A MARSHALL CAVENDISH 7 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00 Republic of Ireland £1.25 Malta 85c Australia $2.25 New Zealand $2.95

INPUT

Vol 1 No 7

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front cover, Jon Couzins. Pages 193, 195, 196, 199, Paddy Mounter. Pages 198, 200, Ray Duns. Pages 201, 202, 203, 204, 205, 206, 207, Tudor Art Studios. Page 208, Jon Couzins. Page 210, Tony Roberts. Pages 210, 211, 213, 214, 215, Bernard Fallon. Page 220, Nick Mijnheer. Page 222, Howard Kingsnorth. Joysticks courtesy of Sonic Foto and Micro Center & Lion House, both of Tottenham Court Road, London W1.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.



HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland: Send £4.95 (inc p & p) (IR£5.45) for each binder to the address below:
Marshall Cavendish Services Ltd, Department 980, Newtown Road, Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Gordon and Gotch Ltd, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Copies of any part of INPUT can be obtained from the following addresses at the regular cover price, with no extra charge for postage and packing:

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services, Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent

COPIES BY POST

Our Subscription Department can supply your copies direct to you regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. These rates apply anywhere in the world. Send your order, with payment to:
Subscription Department, Marshall Cavendish Services Ltd, Newtown Road, Hove, Sussex BN3 7DN
Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – **and please do not telephone.** Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +
COMMODORE 64 and 128
ACORN ELECTRON, BBC B and B+
DRAGON 32 and 64
ZX81
VIC 20
TANDY TRS80 COLOUR COMPUTER

CREATING LEVELS OF DIFFICULTY

- HOW TO DRAW A RANDOM MAZE
- TWO WAYS TO MAKE THE GAME MORE DIFFICULT
- HOW TO MOVE THE MAN
- ADDING UP THE SCORE



Some maze games are very easy to solve but this one gives you two levels of difficulty *and* draws a different maze each time. See how long *you* take to find the treasure

Computer games often ask you to choose a level of difficulty before you start playing. This allows both beginners and experts to play the same game, without it being too difficult or too easy for either.

There are many ways you can introduce levels of difficulty, depending on the exact nature of the game. For example, you could change the number of enemies, introduce a range of delays into the game, allow more or less time, vary the problems, and so on.

This time in Games Programming you'll see how levels of difficulty can be incorporated into a maze game. The game uses one of two ways to generate different levels of difficulty—which method you will see in detail depends on which machine you have. The game doesn't involve just trying to find a way through the maze, but the player has a fixed time limit in which to guide a man to some treasure plotted somewhere in the maze. Note that there is no version for the ZX81 computer.

To make the task easier or more difficult, there are two possible methods you could use. You could vary the complexity of the maze itself, or you could alter the time limit.

The reason why different methods were chosen for different computers is to do with the way in which the maze is generated on them. This is a good example which shows that when you want to devise games with different levels of difficulty you'll have to choose which route you want to follow.

LIVES

When the time runs out for the player trying to find the treasure, you will want to impose some kind of penalty. You could make the player lose some of his score, but the most widely used penalty is to make him lose a life.

In this game the player is given three lives, so if you fail to find the treasure within the time limit on three occasions, the game ends.

RANDOM MAZES

The maze game is based on a random maze-generating subroutine, which is an interesting program in itself because it draws a different maze every time—saving you having to design a whole series of mazes. Remember how on page 68 you were shown how to design a maze and use DATA lines to incorporate it into a program, and then imagine how complicated it would be to devise a whole series.

Designing random mazes is much easier than that, but more complicated than you might imagine. An obvious way of designing them might be to print a number of blocks, say ROM graphics, randomly on the screen. But the problem is that you might not end up with a maze at all, because there is no guarantee of a route through the maze and to use this method you would have to devise some way of checking for a way out.

HOW TO PLOT RANDOM MAZES

The best way of drawing random mazes is to devise a program which plots a random path and builds it up into a maze. The program for your machine is designed so that the line is contained within a frame drawn on the screen. The line isn't allowed to cross over itself, either. When the random path cannot go any further—either it gets stuck in a corner, or between itself and the frame, or it may even get caught up in itself—the computer retraces its steps. It does this a step at a time and examines the area around the path for clear space. When the machine finds a space, another branch of the random path is started, then continued until it is stuck again, and its steps are once more retraced. The computer keeps trying to draw new branches until the frame is filled—when it gets back to where it started.

After the program has finished drawing the maze, there is only one way through—this way can seem quite obvious because the branches of the route aren't complicated. The maze is also solvable by the 'right hand rule'—following the right hand (or left hand, for that matter) wall of the maze all the time. To stop someone doing this you need 'islands' in the maze to break up the walls. So, after drawing the maze, the program then plots a number of random blocks which make the maze seem more complex and will thwart anyone using the right hand rule.

SAVE the program on tape once you've entered it all in because the next article shows how to add some sound effects.

The Spectrum program begins by setting up UDGs, initializing variables and generally preparing for the game. Type in this section of program, but don't RUN it yet:

```
10 FOR n=0 TO 23: READ a: POKE USR
   "a"+n,a: NEXT n
20 LET hs=0
30 INPUT "Select level (1 to 6)□";ta
40 LET ta=1100-100*ta
50 BORDER 1: PAPER 1: INK 0: CLS :
   INK 7
60 LET s=0: LET lives=3
70 PRINT BRIGHT 1; PAPER 6; INK 2;
   "□SCORE□□□□□□□□HIGH
   SCORE□□□□□□□□□"
490 DATA 24,24,60,82,82,24,36,36,127,
   65,93,85,81,95,64,127,24,24,255,
   255,24,24,24,24
```

Line 10 sets up the UDGs for the game—a man, some treasure, and a cross—by READing the DATA in Line 490. The high score—hs—is set at 0 by Line 20.

Next, the player is asked to choose a level of difficulty from 1 to 6—the lower the number, the easier the level. You'll see that in Line 40 lower numbers set longer times, and higher numbers shorter times.

The display colours are set by Line 50, before Line 60 initializes the score to 0 and lives to 3. Finally, Line 70 displays the words SCORE and HIGH SCORE, along with spaces for numbers on the screen.

DRAWING THE MAZE

Now type in these lines:

```
80 FOR n=22561 TO 22589: POKE n,16:
   POKE n+640,16: NEXT n
90 FOR n=1 TO 21: POKE 22528+
   n*32,16:POKE 22558+n*32,16:
   POKE 22559+n*32,9:NEXT n
100 LET b=22593: LET a=b
110 DIM a(4): LET a(1)=-1: LET a(2)
   =-32: LET a(3)=1: LET a(4)=32
120 POKE a,56
130 LET j=INT (RND*4)+1: LET g=j
140 LET b=a+a(j)*2: IF PEEK b=8 THEN
   POKE b,j: POKE a+a(j),56: LET
   a=b: GOTO 130
150 LET j=j+1: IF j=5 THEN LET j=1
160 IF j<>g THEN GOTO 140
170 LET j=PEEK a: POKE a,56: IF j<5
   THEN LET a=a-a(j)*2: GOTO 130
180 POKE 22625,56
190 FOR n=1 TO 20
200 LET k=22528+64*(INT (RND*9)+2)
   +INT (RND*29)+1
210 POKE k,56: NEXT n
```

The border of the maze is set up by Lines 80 and 90—POKEing 16 into the attributes area of memory sets the PAPER colour to red, so you have a border consisting of red blocks.

Lines 100 to 180 are the maze drawing lines. Don't be tempted to [BREAK] the program and clear the screen—the maze will be lost because it is only stored in the attributes file.

To complete the maze, Lines 190 to 210 display 20 squares at random positions within the maze. If the squares land on a wall they change it into pathway.

CREATING A GAME

The next section of program deals with the game itself:

```
220 LET x=15: LET y=10
230 LET tx=INT (RND*15)*2+1
240 LET ty=INT (RND*10)*2+2
250 PRINT BRIGHT 1; PAPER 2;AT 0,7;
   s;AT 0,24;hs
260 POKE 23672,0: POKE 23673,0
270 PRINT FLASH 1; PAPER 3; INK 6;AT
   ty,tx;CHR$ 145
280 PRINT INK 2; PAPER 7;AT y,x;CHR$
   144
290 IF PEEK 23672+256*PEEK 23673>ta
   THEN GOTO 390
300 IF INKEY$="" THEN GOTO 290
310 LET a$=INKEY$: LET sx=x: LET sy=y
320 IF a$="z" AND ATTR (y,x-1)>=56
   THEN LET x=x-1
330 IF a$="x" AND ATTR (y,x+1)>=56
   THEN LET x=x+1
340 IF a$="k" AND ATTR (y-1,x)>=56
   THEN LET y=y-1
350 IF a$="m" AND ATTR (y+1, x)>=56
   THEN LET y=y+1
360 PRINT PAPER 7; INK 2;AT sy,sx;
   "□";AT y,x;CHR$ 144
370 IF ty=y AND tx=x THEN GOTO 470
380 GOTO 290
```

The starting position of the man is set by Line 220—the man starts at 15,10 at the beginning of each game.

The treasure is placed randomly in the maze by Lines 230 and 240 and Line 270 displays it on screen. The range of random numbers chosen in Lines 230 and 240 guarantee that the treasure will land on a path.

Lines 250 displays values for SCORE and HIGH SCORE—initially 0—and Line 260 sets the clock to 0 by POKEing two memory locations as you saw on page 101. Line 290 checks if the time limit has been exceeded, and if it has, jumps to line 390.

The man is displayed by Line 280. Notice that in Lines 270 and 280 CHR$ 144 and

CHR$ 145 are used to display the UDGs on the screen. CHR$ has been used here because it is the clearest in program listings, although you could use the alternative method using letters as explained on page 44.

The remaining lines—from Line 300 to Line 380—deal with moving the man about the maze. Lines 320 to 350 check that the appropriate key is being pressed and that the next square is path not wall before moving him. The check uses ATTR which was fully explained for the simple maze game on page 69. Line 260 blanks out the man's last position and displays him at his new one.

Line 370 tests if the man has reached the treasure. If he has, it increases the score before displaying another treasure for the man to search out.

RAISING THE DEAD

The final section of program is below. At last you can RUN the game.

```
390 PRINT FLASH 1; PAPER 0; INK 5;
    AT y,x;CHR$ 146
400 LET lives=lives-1: FOR f=1 TO 200:
    NEXT f: IF lives>0 THEN GOTO 260
410 IF s>hs THEN LET hs=s
420 PRINT BRIGHT 1; PAPER 2; AT 0, 24;hs
430 PRINT FLASH 1;AT 10,1;"□PRESS
    ANY KEY TO PLAY AGAIN□"
440 IF INKEY$<>"" THEN GOTO 440
450 IF INKEY$="" THEN GOTO 450
460 GOTO 30
470 LET s=s+ta-PEEK 23672+256*PEEK
    23673: GOTO 230
```

Line 390 displays a flashing cross (CHR$ 146) when the player loses a life before Line 400 subtracts 1 from the lives total. There is a pause before the game returns to Line 260 which resets the clock ready for another attempt to reach the treasure. Of course, it only returns if there are lives left.

If no lives remain, Line 410 compares score and high score. High score is altered if the latest score is higher. In either case, Line 420 displays the high score.

Line 440 is needed in case the player is still pressing one of the move keys before Line 450 checks for a keypress to signal another go.

Finally, Line 470 calculates the score after the man and treasure coincide. The check, you may remember, was in Line 370.

C

When this program is RUN, you are asked for a level of difficulty. 1, 2, 3, or 4 can be entered and these correspond to playing times of 20, 16, 12 and 8 seconds.

The object of the game is to reach the treasure—a randomly positioned asterisk—in the shortest possible time. Your 'man' is a pi character and the Z, X, P and L keys are used to direct him to the target:

```
50 POKE53280,6:INPUT"■■INPUT LEVEL
   (1-4) ■";A:IFA<1ORA>4THEN50
60 LE=5-A:LE=LE*4+4
100 PRINT"■■■":A=1186:POKE
    650,255
105 FORZ=0TO39:POKE1104+Z,102:POKE
    1984+Z,102:POKE55376+Z,1:POKE
    56256+Z,1:NEXT
110 A(0)=-1:A(1)=-40:A(2)=+1:A(3)
    =40:FOR F=1 TO 21
150 PRINT"■▒■π□□□□□□□
    □□□□□□□□□□□□□□□□□□
    □□□□□□□□□□□□□□□□□□
    □■■▒";:NEXT F:POKEA,4
220 J=INT(RND(1)*4):G=J:POKE54272+A,7
230 B=A+A(J)*2:IFPEEK(B)=160THEN
    POKEB,J:POKE54272+A,6:POKE
    A+A(J),32:A=B:GOTO220
240 J=(J+1)*-(J<3):IFJ<>GTHEN230
250 J=PEEK(A):POKEA,32:IFJ<4THEN
    A=A-A(J)*2:GOTO 220
1000 LV=3:FORZ=1TO60:X=INT(RND(1)*
     40+1)+INT(RND(1)*9)*80
1002 IFPEEK(1184+X)=160ANDPEEK
     (1224+X)=160THENPOKE1184+X,32
1004 NEXT Z
1006 X=RND(1)*720:IFPEEK(1223+X)<>
     32THEN1006
```



```
1008 OL=1223+X:POKEOL,42
1010 TI$="000000":POKEA,94:J=3
1013 PRINT"■";:FORZ=1TO62:PRINT
    "□";:NEXT:PRINT"■HIGH
    SCORE:■"HS
1014 PRINT"■■LV:■"LV"□□□□
    ■TIME:■"TI$"□□□□□□
    ■SCORE:■"SC
1015 IF VAL (TI$)>=LE THEN 2000
1016 GETZ$:IFZ$=""THEN1014
1017 IF Z$="Z"THENJ=0
1018 IF Z$="X"THENJ=2
1019 IF Z$="P"THENJ=1
1020 IF Z$="L"THENJ=3
1021 B=A+A(J):IFPEEK(B)<>102 AND
    PEEK(B)<>160 THEN 1040
1030 GOTO 1014
1040 IFPEEK(B)=42THEN3000
1050 POKEB,94:POKEA,32:A=B:GOTO 1014
2000 LV=LV-1:FORZ=155TO0STEP-1:
    POKEA,RND(1)*6+109
2003 NEXT:POKEA,94:IFLV>0THEN1010
2005 PRINT"■□□□■F1□■-□
    NEW MAZE□□□□□□□□□□□□□
    ■■■■■■■■■■■■■
    ■□□□■F7□■-□□START"
2006 SC=0:LV=3:GETK$:IFK$="■"
    THEN50
2007 IFK$="■"THEN1010
2010 GOTO2006
3000 SC=SC+50-VAL(TI$):POKEOL,32:
    IFSC>HSTHENHS=SC
3010 GOTO1006
```

On the Vic 20, you need to use these Lines in place of those with the same Line numbers in the program for the Commodore 64:

```
50 POKE 36879,110:INPUT"■■
    INPUT LEVEL(1-4)■";A:IFA<1 OR
    A>4THEN50
100 PRINT "■■■":A=7770:POKE
    650,128
105 FORZ=0TO21:POKE7724+Z,102:
    POKE8142+Z,102:POKE38480+Z,1:
    POKE38862+Z,1:NEXT
110 A(0)=-1:A(1)=-22:A(2)=1:
    A(3)=22:FORF=1TO18
150 PRINT "■■■π□□□□□□□
    □□□□□□□□□□□□□□□□
    □■■■";:
    NEXT F:POKEA,4
220 J=INT(RND(1)*4):G=J:POKE
    30720+A,7
230 B=A+A(J)*2:IF PEEK(B)=160
    THEN POKEB,J:POKE30720<A,6:
    POKEA+A(J),32:A=B:GOTO220
1000 LV=3:FORZ=1TO60:X=INT(RND
    (1)*22)+1+INT(RND(1)*8)*44
1002 IFPEEK(7724+X)=160ANDPEEK
    (7746+X)=160THENPOKE7724+X,32
1006 X=RND(1)*396:IFPEEK(7724+X)
    <>32THEN1006
1008 OL=7724+X:POKEOL,42
1013 PRINT"■";:FORZ=1TO33:
    PRINT"□";:NEXT:PRINT"■HS:
    ■";HS
1015 PRINT "■■LV:■"LV
    "□□□□□□■TIME:■"TI$
    "■SC:■"SC:IFVAL(TI$)>=
    LE□THEN2000
2005 PRINT"■□■F1■-NEW
    MAZE□■F7■-START□"
```

Both of the Commodore programs start by setting the screen colour and asking for the desired playing level (Line 50). The actual routine for creating the maze is contained in Lines 100 to 250. First, a chequered border is drawn and the enclosed area is immediately filled with yellow blanks using the PRINT string of Line 150. The physical limits of this are set in Line 110.

The random pattern of the maze is produced by Line 220, where J can be given any value from 0 to 3. This is later converted to the characters @, A, B and C which represent the directions left, up, right, and down. When the program is RUN and the maze is being created, you can see these characters flashing briefly on the screen. The corresponding directional change is made and a blank is left behind—in the normal screen colour—and it is this that forms part of the maze. The donkey work is carried out by Line 230.

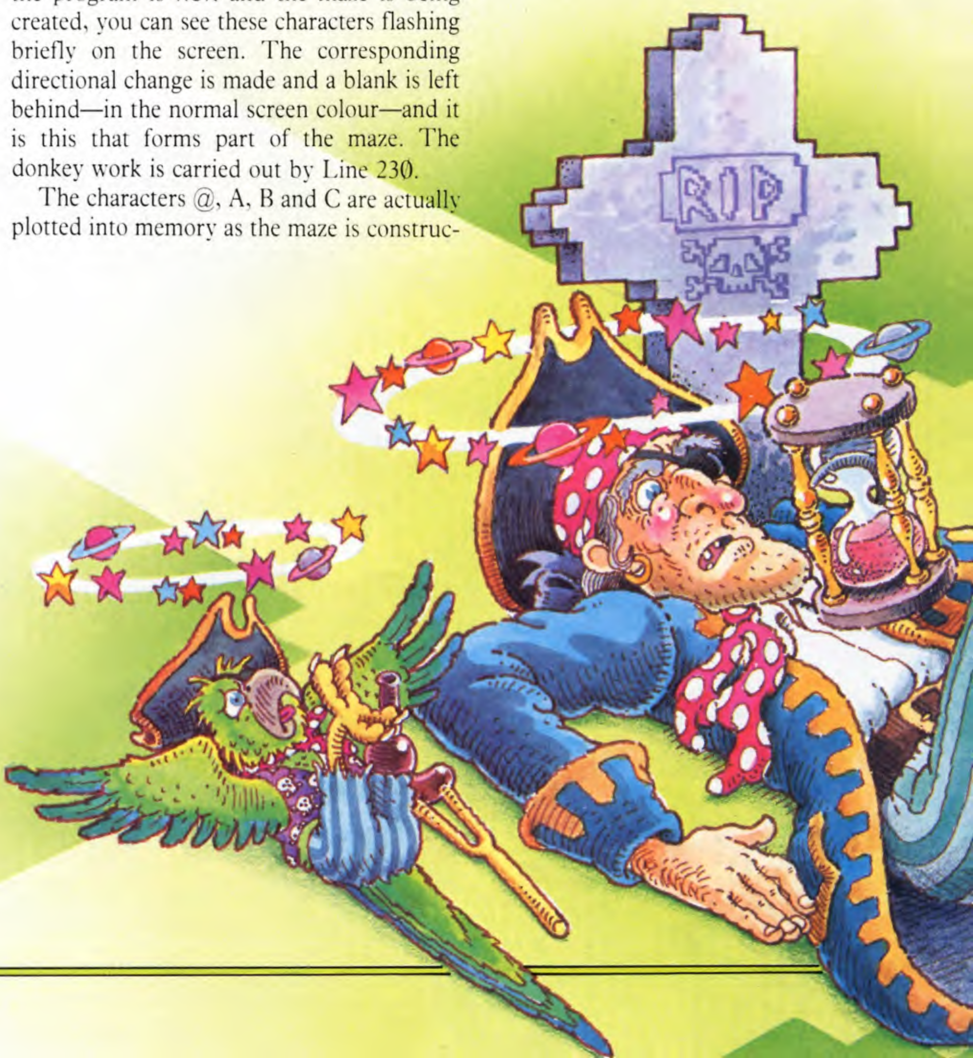
The characters @, A, B and C are actually plotted into memory as the maze is constructed but do not appear on the screen—except momentarily—because they are overprinted with the background colour. When the moving cursor is able to move into a yellow blank square, it does so. If it cannot move in the first randomly chosen direction it tries another, eventually returning along its original route when all the allowable yellow blanks have been removed. The cursor tries at every point to go anywhere *but* back along its original route, and this is how areas of yellow blanks which were originally bypassed are eventually incorporated within the maze.

The last part of Line 240 checks to see if the cursor has reached its home position, the starting point. If it hasn't, the program jumps to the next line which resets the direction and maze-making routine.

Lines 1000 to 1004 then print further blanks, at random along the maze wall.

Line 1006 then randomly selects the position of the asterisk (the treasure), which is printed using the POKE in the following line.

The program then sets the built-in timer to zero, clears anything from the top rows (where later a screen prompt is displayed) and also PRINTs the high score display (variable HS), lives remaining (LV-1), time elapsed



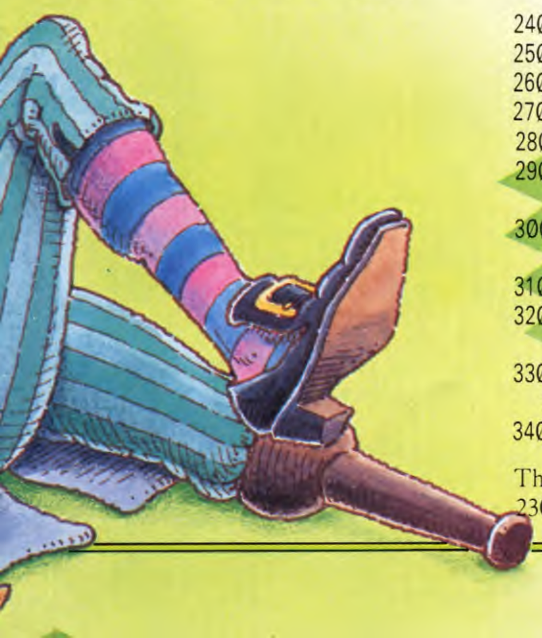
since the built in clock was zeroed (TI), and the current score (SC). The final part of Line 1015 checks whether or not your time is up.

The next five lines of the program test for your keypress, entering the corresponding value into the first part of Line 1021, your new location. The PEEKs which follow this check to see if the direction is valid, and that you haven't gone into the maze wall. The routine at Lines 1040 and 1050 checks whether you have hit the asterisk and, if you haven't, prints your man at his new position as well as deleting him at his former position. The new location is set (A=B) and the RETURN is made. The next line (1030) re-directs the program to Line 1015 which updates the score and checks the time.

If the check proves negative the GOTO directs the program to Line 2000 where the lives count (LV) is reduced by 1 and your man flickers through a change of 155 characters. Line 2003 then replaces your man, restarting the program again at Line 1010 if you have lives remaining, displaying screen prompts for a new maze and restart if you do not. Line 2006 resets the score and lives variables and checks for an [F1] keypress, in effect reRUNning the program if this is detected. Line 2007 responds to an [F7] keypress by restarting the game part of the program.

If, during a game, you manage to reach the asterisk within the time limit, the routine at Lines 3000 and 3010 adds the updated score, deletes the asterisk and sets a new high score.

The Acorn computers use a simple change of MODE to create the two levels of difficulty. Since MODE 0 has twice as many columns as MODE 1 this is a straightforward way of making the maze twice as difficult. You are asked to choose between an easy maze in MODE 1, and a harder one in MODE 0.



Here is the main part of the program. The procedures are all defined later on but their names have been chosen so you know straightaway what each does. The variables used at this stage are: D, the level of difficulty; X and Y, the start position of the man; M, the number of men or lives and SC, your score.

```
10 PROCinitialize
20 MODE(1-D)
30 PROCconstructmaze
40 PROCdisplaymaze
50 TIME=0
60 X=32/(D+1):Y=1023-4*32
70 VDU5
80 REPEAT
90 PROCplay
100 UNTIL M<0
110 PROCend
120 MODE1
130 PRINTTAB(15,15)"SCORE□";SC
140 PRINTTAB(15,20)"PRESS RETURN"
150 PRINTTAB(15,21)"TO PLAY AGAIN"
160 *FX21
170 IF INKEY(-74) THEN RUN
180 GOTO 170
190 END
```

Using procedures like this makes the structure of the game very easy to understand. Unfortunately, it is not possible to change MODE inside a procedure so Line 20 MODE(1-D) has to stand separately. At the end too, it would have been neater to include Lines 120 to 180 in PROCend, but the need to change MODE before printing out the instructions again meant this was impossible.

The first procedure—PROCinitialize—sets up all the variables and defines the UDGs:

```
200 DEFPROCinitialize
210 *TV254,1
220 DIM S(3)
230 PRINT"DIFFICULTY 0-EASY 1-NOT
    SO EASY ?"
240 D=GET-48
250 IF D<>1 AND D<>0 THEN 240
260 M=2:SC=0:MT=3000
270 *FX11,10
280 *FX12,10
290 VDU23,255,255,255,255,255,255,
    255,255,255
300 VDU23,224,255,255,255,255,255,
    255,255,255
310 VDU23,225,24,24,60,90,90,24,36,36
320 VDU23,226,0,255,129,189,165,173,
    161,191
330 VDU23,227,24,24,255,255,24,24,
    24,24
340 ENDPROC
```

The level of difficulty is sorted out in Lines 230 to 250. Line 240 GETs the ASCII code of the number you type in. But the ASCII value of 0 is 48 and the value of 1 is 49, so you have to take away 48 to get back to 0 and 1 again. If you type in anything other than 0 or 1, Line 250 sends you back for another go. Line 210 moves the TV display down one line, Line 220 dimensions the array S which makes sure there is always a wall separating parallel paths. Line 260 gives you 3 lives (from 0 to 2), a zero score and a maximum time of 30 seconds to find each treasure. Note that on the Electron this time may be too short so change it to MT=5000. The *FX calls in Lines 270 and 280 speed up the auto repeat then the next five lines define the UDGs.

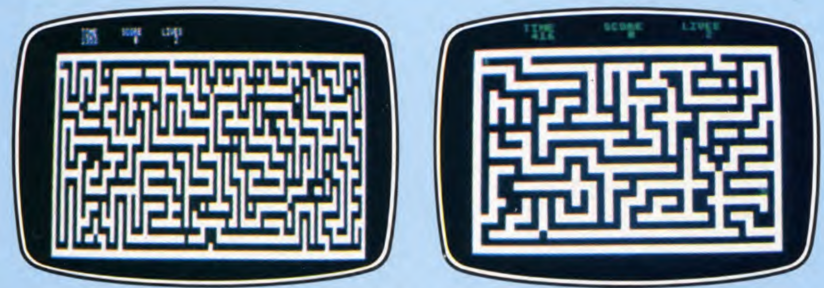
Characters 224 and 255 are both solid blocks. Character 224 is used for the main part of the maze and 255 for the boundary wall. Although they appear the same on the screen, the computer can tell the difference and can make sure that the pathway won't cross the boundary. The other characters are the man—225, the treasure—226 and a cross—227 that appears when you lose a life.

DRAWING THE MAZE

The next procedure in the program constructs the maze in the computers memory:

```
350 DEFPROCconstructmaze
360 VDU23;8202;0;0;0;
370 A=&7000+41+40*D
380 S(0)=-1:S(1)=-40-D*40:S(2)=1:
    S(3)=40+D*40
390 FOR Z=0 TO (38+D*40):?(&7000+Z)
    =224:?(&7000+Z+960+D*960)
    =224:NEXT
400 FOR F=1 TO 23:?(F*(40+D*40)+
    &7000)=224:?((F+1)*(40+D*40)
    +&7000-2)=224
410 FOR T=1 TO 37+D*40:?(F*(40+
    D*40)    +&7000+T)=255:NEXT:NEXT
420 ?A=4
430 J=RND(4)-1:G=J
440 B=A+S(J)*2:IF ?B=255 THEN ?B=J:
    ?(A+S(J))=32:A=B:GOTO 430
450 J=(J+1)*-(J<3):IF J<>G THEN
    440
460 J=?A:?A=32:IF J<4 THEN A=
    A-S(J) *2:GOTO 430
470 ENDPROC
```

The area of memory chosen is part of the memory used for the screen so you can actually see the maze being built up. (Although it doesn't look much like a maze until later.) Line 370 sets the start address in the memory and Line 380 calculates the amount to step in each direction. Lines 390 and 400 fill the boundary of the maze with character 224 and Line 440 fills the whole of the centre area with 255.



Two levels of difficulty on the Acorn: hard and rather easier

The other lines create the pathway through the maze. The path takes a random direction checking memory locations all the time to make sure no part of the maze is missed. Wherever the path goes, the value 32—for a space—is put in the memory location in place of the block character 255.

PROCdisplaymaze then peeks all the locations to find the value, W, stored there and prints out a white block if W equals 224 or 255 and a space if W equals 32; in other words, it prints out the maze itself:

```
480 DEFPROCdisplaymaze
490 VDU19,1,2,Ø,Ø,Ø
500 VDU1Ø,1Ø,1Ø
510 FOR Y=Ø TO 24
520 FOR X=Ø TO 38+D*4Ø
530 W=?(&7ØØØ+X+Y*(4Ø+D*4Ø))
540 VDU W
550 NEXT
560 VDU32
570 NEXT
580 PRINTSTRING$(4Ø+4Ø*D,"□")
590 FOR T=1 TO 5Ø+D*5Ø
600 X=RND(36+D*4Ø)+1:Y=RND
    (23)+3
610 PRINTTAB(X,Y)"□"
620 NEXT
630 COLOUR1
640 VDU31,X,Y,226
650 PRINTTAB(6,Ø)"TIME","SCORE",
    "LIVES"
660 ENDPROC
```

The maze is printed out by Lines 51Ø to 58Ø. The next four lines print a few random spaces to break up the walls and create a few islands in the maze. Then Line 64Ø prints the treasure and Line 65Ø prints the headings for the time, score and number of lives.

Notice how D, the level of difficulty is included in all the calculations so the maze will fit either a 4Ø or 8Ø column screen.

PLAYING THE GAME

Now type in the next section which controls how the game is played. DEFPROCplay moves the man and times how long it takes to reach the treasure:

```
670 DEFPROCplay
680 MOVE X,Y:GCOLØ,Ø:VDU224,8:GCOL
    Ø,1:VDU225
690 LX=X:LY=Y
700 *FX21
710 K$=INKEY$(1)
720 VDU4:PRINT'TAB(Ø,1)TIME,SC,M:
    VDU5
730 IF TIME>MT THEN PROCloselife
740 IF M<Ø THEN ENDPROC
750 IF K$="" THEN 71Ø
760 IF K$="Z" AND POINT(X-32+D*16,
    Y)<>3-D*2 THEN X=X-32+D*16
770 IF K$="X" AND POINT(X+32-D*16,
    Y)<>3-D*2 THEN X=X+32-D*16
780 IF K$="L" AND POINT(X,Y-32)<>3
    -D*2 THEN Y=Y-32
790 IF K$="P" AND POINT(X,Y+32)<>3
    -D*2 THEN Y=Y+32
800 VDU4
810 VDU31,X/(32-16*D),32-Y/32
820 A%=&87:H=(USR(&FFF4) AND
    &FFØØ)/&1ØØ : H=H+96
830 IF H=226 THEN PRINTTAB(RND(36
    +D*4Ø)+1,RND(22)+4)CHR$(226):
    SC=SC+MT-TIME: TIME=Ø
840 VDU5
850 MOVE LX,LY:GCOLØ,Ø:VDU224
860 ENDPROC
```

Most of this is very similar to the routines given in Games Programming 3 and a lot of the lines should be quite familiar by now. Line 68Ø moves to the start position of the man, blanks out the background then prints a green man. Lines 71Ø and 75Ø to 79Ø find out what key you're pressing so they can calculate the new position, then Line 81Ø moves the cursor to that position. Line 82Ø uses an operating system routine to find out what character is in that square, storing the result in H. If H=226—that is, you've found the treasure—then a new treasure is printed at a new random position, your score is increased and the time is reset. You then have another 30 seconds to find the new treasure and so on.

If your time ever runs out before you find the treasure, then Line 73Ø comes into operation and you lose a life.

```
870 DEFPROCloselife
880 M=M-1
890 VDU4:PRINTTAB(X/(32-16*D),
    32-Y/32)CHR$(227);
900 FOR DE=1 TO 2ØØØ:NEXT
910 VDU8,225
920 VDU5:TIME=Ø
930 ENDPROC
```

This procedure reduces your lives by one. Then, to show your man is dead, Line 89Ø prints a cross and there is a short delay before Line 91Ø prints your next man.

When you've lost all three lives, PROCend is called which blanks out the man and resets the autorepeat and the cursor:

```
940 DEFPROCend
950 MOVE X,Y:GCOLØ,Ø:VDU255
960 *FX12,Ø
970 VDU 4
980 FOR DE=1 TO 2ØØØ:NEXT
990 ENDPROC
```

Finally, the main program displays your final score and gives you the choice of another go.

On the Dragon and Tandy, the easiest way to draw the random mazes would be to use the text screen. Unfortunately, because of the large size and small number of the blocks available, the mazes would be too simple.

Instead, the program draws the mazes on the high resolution graphics screen. Although the program is more complex than one designed to draw on the text screen, it has the added advantage that mazes of a range of different complexities can be drawn, giving a range of different levels of difficulty.

Imagine that the random path consists of a series of square blocks. If you wanted to draw a simple maze you would choose a large block size, but if you wanted a more complex maze, you would choose a small block size.

The first section of the program initializes the variables and generally prepares the computer for drawing random mazes. Type it in, but don't RUN it yet because you'll get a UL—Undefined Line—error when the program tries to GOSUB 1ØØØ.

```
10 PMODE4,1
20 CLS:PRINT@193," LEVEL OF
   DIFFICULTY (Ø-5) ";
30 L$=INKEY$:IFL$<"Ø"ORL$>"5"
   THEN3Ø
40 BS=12-VAL(L$):NX=2*INT(.5+
   128/BS):NY=2*INT(.5+96/BS)
```

```
50 SX=25Ø-BS*NX:SY=19Ø-BS*NY
6Ø DIMP(NX,NY),A(5),B(5)
7Ø PCLS5:DRAW"S"+STR$(INT(8.5-
   4*VAL(L$)/5))+"CØBMØ,ØBR2BD
   NFNGD3NFG"
8Ø GET(Ø,Ø)-(BS-1,BS-1),A,G
9Ø GET(1Ø,1Ø)-(BS+9,BS+9),B,G:
   COLOR5,Ø
1ØØ CLS:PRINT@228,"GENERATING
   LEVEL□";L$;"□MAZE"
11Ø GOSUB1ØØØ
12Ø GOTO 12Ø
```

Line 1Ø tells the computer that you will want to use PMODE 4 during the program. The high resolution screen isn't switched on at this stage, so you will still be looking at the text screen. Line 2Ø then displays the message LEVELS OF DIFFICULTY (Ø-5).

L$ is the level that the player has chosen. The numeric value of L$—VAL(L$) in Line 4Ø—regulates the block size, and hence the width of the path and the complexity of the maze. INKEY$ in Line 3Ø means that the player cannot type more than a single digit before the program continues and saves the player from tapping [ENTER].

In Line 4Ø BS is the block size in pixels—the size can vary from 7 to 12 pixels. NX is the number of blocks along the screen, and NY is the number of blocks up the side.

Before the computer draws the maze on the screen, it works out what it will look like and feeds the information into array P, which is DIMensioned in Line 6Ø. Array A will contain the shape of a man, and B a blank for animating him.

The man is DRAWn by Line 7Ø. On page 185 you saw how you can DRAW pictures in a range of sizes. In this program the man is DRAWn larger when the paths are wider, and smaller when the paths are narrower.

Now that the man has been DRAWn Line 8Ø GETs him into array A, and Line 9Ø fills array B with white. Previously, when you've seen a blank used in a game you haven't had to GET anything into the array—it was simply an empty array. This time, though, you want a blank that is white to match the path.

The COLOR command in Line 9Ø is there so that the maze is drawn in the correct colour later in the program—otherwise you'd be drawing a black maze on a black background!

Line 1ØØ tells the player that the maze is being generated—it can take some time for the maze to appear.



DRAWING THE MAZE

Now type in the maze-generating subroutine—called by Line 11Ø—and you can RUN the program:

```
1ØØØ FORJ=ØTO NX:P(J,NY)=6:P(J,Ø)
   =6:NEXT
1Ø1Ø FORJ=ØTO NY-2:P(Ø,J)=6:P(NX,
   J)=6:NEXT
1Ø2Ø X=2:Y=2:LX=2:LY=2
1Ø3Ø J=RND(4)-1:G=J
1Ø4Ø Y=LY+2*((J=Ø)-(J=2)):X=
   LX+2*((J=3)-(J=1))
1Ø5Ø IFP(X,Y)=Ø THENP(X,Y)=J+1:
   P((X+LX)/2,(Y+LY)/2)=5:LX=X:
   LY=Y:GOTO1Ø3Ø
1Ø6Ø J=(J+1)AND3:IF J<>G THEN1Ø4Ø
1Ø7Ø J=P(LX,LY)-1:P(LX,LY)=5:IF
   J<4 THEN LX=LX-2*((J=3)-
       (J=1)):LY=LY-2*((J=Ø)-
       (J=2)):GOTO1Ø3Ø
1Ø8Ø FORJ=ØTO2Ø:P(2+2*RND
   ((NX-3)/2),1+RND(NY-3))=5:
   P(1+RND(NX-3),2+2*RND
   ((NY-3)/2))=5:NEXT
1Ø9Ø SCREEN 1,1:PCLS
11ØØ FORJ=2TO NX-2:FORK=2TO
   NY-2
111Ø IFP(J,K)=5 THEN LINE
   (J*BS+SX,K*BS+SY)-((J+1)*
   BS+SX-1,(K+1)*BS+SY-1),
   PSET,BF
112Ø NEXTK,J:RETURN
```

Lines 1ØØØ to 1Ø8Ø 'draw' the maze in the computer's memory, and store the shape in the array P—each element in the array corresponding to a block in the maze. The subroutine stores the number 5 in P wherever there's a path, and leaves a Ø if there's a wall.

Once the maze has been stored in P, Line 1Ø9Ø switches on the high resolution screen ready for the maze to be drawn.

Lines 1100 to 1120 display the maze on the screen by examining the contents of P. Whenever a 5 is found, a white square is printed.

MOVEMENT, TREASURE, LIVES

You now need a game to go with your random mazes. Type in the next section of program, but again, don't RUN it because it calls a subroutine which doesn't yet exist. On the Tandy use 251 in Line 170 instead of 239; 253 instead of 247 in Line 180; 247 instead of 223 in Lines 190 and 200:

```
120 X=2:Y=2:LX=2:LY=2:TI=800:LI=3
130 TIMER=0
140 X1=1+RND(NX-3):Y1=1+RND
    (NY-3):IFP(X1,Y1)=5THENP(X1,Y1)=7:
    DRAW"S4C0BM"+STR$(SX+X1*BS)
    +","+STR$(SY+Y1*BS)+"BFR5D5
    L3U3RD" ELSE140
150 X1=X*BS+SX:Y1=Y*BS+SY
160 PUT(X1,Y1)-(X1+BS-1,Y1+BS-1),
    A,PSET
170 IFPEEK(338)=239 THEN Y=Y-1
180 IFPEEK(342)=247 THEN Y=Y+1
190 IFPEEK(340)=223 THEN X=X-1
200 IFPEEK(338)=223 THEN X=X+1
210 IFP(X,Y)=7 THENF=1:P(X,Y)=5:
    GOTO230
220 IFP(X,Y)<>5 THEN X=LX:Y=LY:
    GOTO170
230 IF X<>LX OR Y<>LY THENPUT
    (X1,Y1)-(X1+BS-1,Y1+BS-1),B,
    PSET:LX=X:LY=Y:FORP=1TO68:
    NEXT
240 IFF=1 THENF=0:SC=SC+
    (TI-TIMER):TI=TI-10:GOTO130
250 IFTIMER>TI THENGOSUB500:IFLI<1
    THEN100
260 GOTO150
```

Line 120, which over-writes the previous Line 120, contains variables from which the man's position is calculated. X,Y is the current position of the man, and LX, LY is the last position, but because the path width varies, the values have to be adjusted slightly before the man appears on screen TI is the time limit for recovering the treasure. The time limit is 16 seconds, and if the man hasn't found the treasure when the time expires, he loses a life. At the start of the game the player has three lives—LI=3.

The timer is set to zero in Line 130, before Line 140 selects a position at random for the treasure. The corresponding element in P is examined to make sure that the place is on a path. If it is on a path, the value in the array is changed from 5 to 7. The last part of the line DRAWs the treasure in the maze.

The man's position is calculated in Line 150, taking into account the width of the path which is the block size, BS. Line 160 PUTs the man on the screen at that position.

Lines 170 to 220 should need no explanation by now—they are the keyboard PEEKs which allow you to move the man through the maze. He mustn't be allowed to walk through the walls, though, so Line 220 makes sure that he keeps to the paths. Line 210 checks if the treasure has been found, by examining the corresponding element in P for the number 7. If the computer finds it, then the 'found flag'—F—is set to 1.

Line 230 causes the man to move. The blank is PUT on the man's last position and the current position becomes the last position.

Line 240 calculates the score if the treasure has been found. The time limit is decreased by 10 seconds. F is reset to zero, and the program jumps back to reset the timer. The program continues, replotting the treasure, but leaving the man in the same place as he was when the treasure was found.

If the player takes too long to find the treasure, Line 250 calls the subroutine starting at Line 500. If the man hasn't found the treasure and he still has some time left, Line 260 sends the program back so that his new position is calculated.

DISPLAYING THE SCORE

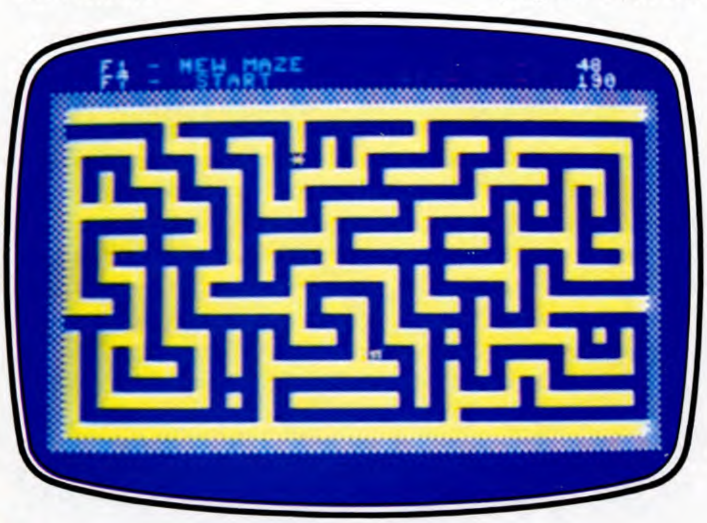
This is the final subroutine. It will display the score and number of lives remaining after a life has been lost:

```
500 CLS:SCREEN0,0:LI=LI-1
510 PRINT@106,"LEVEL = ";L$
520 IFLI>0 THEN PRINT@202,"LIVES = ";LI
530 PRINT@298,"SCORE = ";SC
540 IF LI>0 THENFORJ=1TO6000:NEXT:
    TIMER=0:SCREEN1,1:RETURN
550 PRINT@358,"ANOTHER GAME (Y/N) ?"
560 A$=INKEY$:IFA$<>"Y"ANDA$
    <>"N" THEN560
570 IFA$="Y" THEN RUN
580 END
```

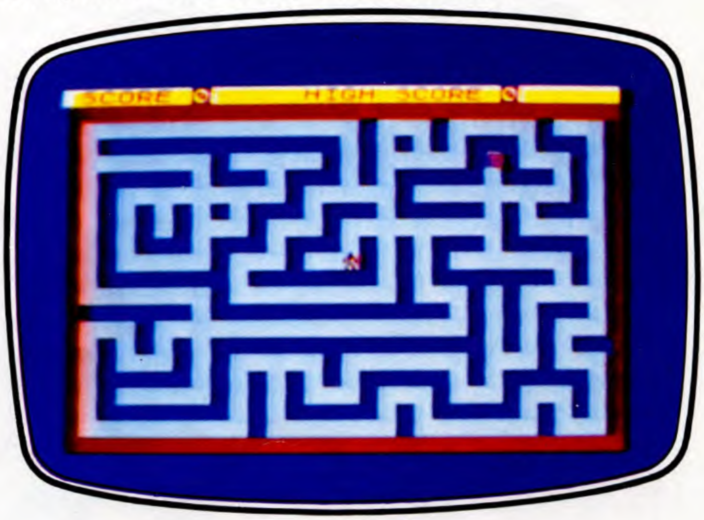
You can now RUN the program. If a life has been lost the program switches back to the text screen—SCREEN 0,0. Line 500 also decreases the number of lives remaining to the player by 1.

Lines 510 to 530 display the level of difficulty, the number of lives remaining—if the game is continuing—and the score. If some lives still remain, Line 540 inserts a pause before resetting the timer, switching back to the high resolution screen and RETURNing from the subroutine.

Lines 550 to 580 ask if the player wants another go, and then either stops the program or reRUNs it. RUN has been used in Line 570 to clear P for the new maze.



A pirate searches the Commodore maze for his booty



The pirate searching the colourful Spectrum maze

UNRAVELLING YOUR STRINGS

- COMPARING AND SORTING STRINGS
- STRING SLICING
- HOW LONG IS A STRING?
- USING STRINGS FOR WORDPROCESSING

Strings are used in all sorts of programs, in fact whenever you want to work with more than just plain numbers. Here are a few ways to make the most of them

A string is made up from a collection of characters. They can be letters, numbers, punctuation or any of the keyboard symbols. And you can put them together in any order you like.

Normally, of course, a string contains useful information—sometimes more than one piece of information in a single string. For example, "PETER WHITE 241Ø67 S" consists of first name, second name, date of birth and marital status—four pieces of information. The date of birth can itself be broken down into day, month, year so there are really six pieces of information in all.

Lots of times you may need to split up (or slice) a string to extract different bits of information—such as the date, in the example above. At other times it may be necessary to add strings together. You may also want to measure the length of a string and work out the value of any numerical parts. All this is possible using a few simple BASIC keywords.

Adding strings—called concatenation—is the easiest of all. You just use the symbol '+'. If A$ equals "HI" and B$ equals " THERE", then A$+B$ equals "HI THERE". Concatenation simply glues strings together; it does not add them. So "439"+"241" equals "439241", not 68Ø.

COMPARING STRINGS

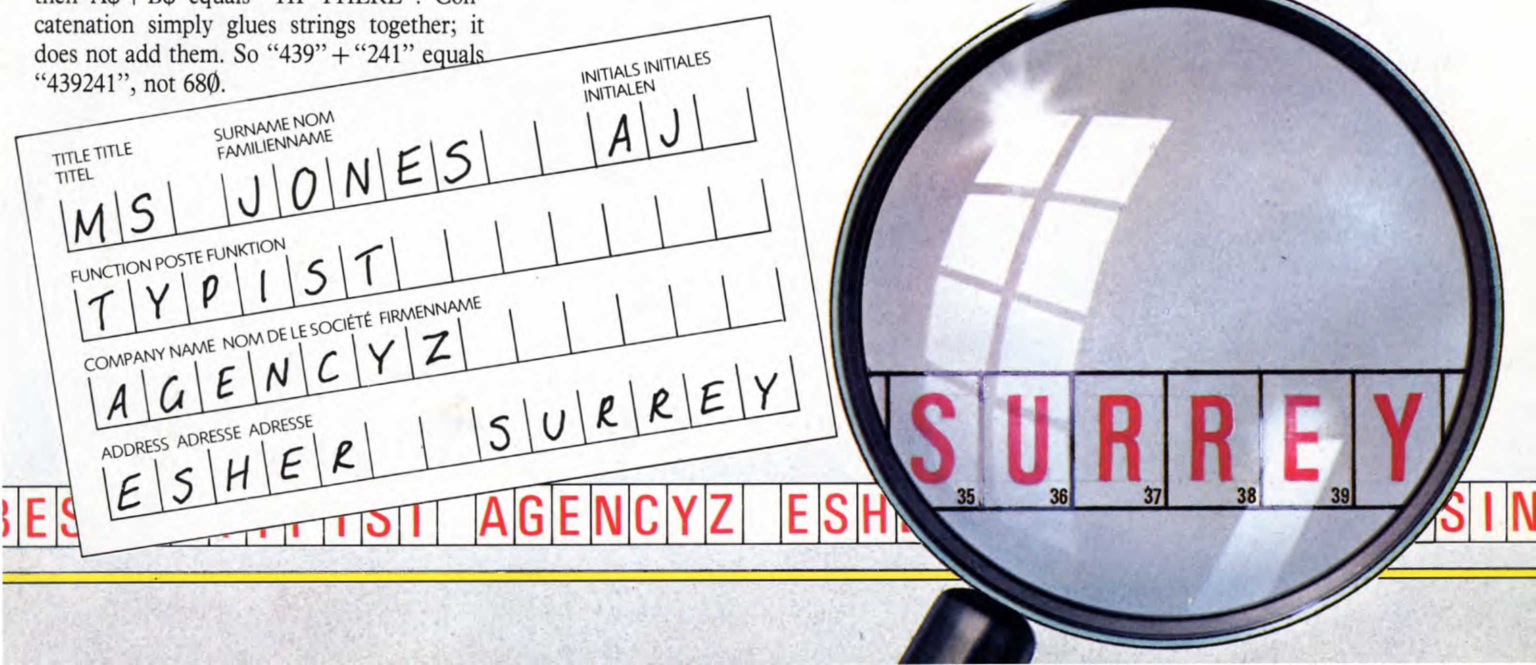
As well as adding strings together, the computer can compare them to see if they are the same, as in this simple guessing game:

```
1Ø G=1: GOTO RND(6)*1Ø+1Ø
2Ø B$="APPLE": GOTO 8Ø
3Ø B$="ORANGE": GOTO 8Ø
4Ø B$="BANANA": GOTO 8Ø
5Ø B$="LEMON": GOTO 8Ø
6Ø B$="PASSION FRUIT": GOTO 8Ø
7Ø B$="PINEAPPLE"
8Ø CLS: PRINT"I AM A FRUIT, WHAT
   FRUIT AM I□";
9Ø INPUT A$
1ØØ IF A$=B$ THEN GOTO 17Ø
11Ø G=G+1
12Ø PRINT"WRONG!"
13Ø FOR J=1 TO 2ØØØ
14Ø NEXT J
15Ø CLS
16Ø GOTO 9Ø
17Ø IF G=1 THEN PRINT "YOU WERE
   RIGHT IN 1 GUESS" ELSE PRINT"YOU
   WERE RIGHT IN□";G;"□GUESSES"
```

This program will work on the ZX81 if you split all the multiple statement lines onto separate lines:

```
1Ø LET G=1: GOTO INT (RND*6)*1Ø+1Ø
2Ø LET B$="APPLE": GOTO 8Ø
3Ø LET B$="ORANGE": GOTO 8Ø
4Ø LET B$="BANANA": GOTO 8Ø
5Ø LET B$="LEMON": GOTO 8Ø
6Ø LET B$="PASSION FRUIT": GOTO 8Ø
7Ø LET B$="PINEAPPLE"
8Ø CLS : PRINT "I AM A FRUIT, WHAT
   FRUIT AM I?"
9Ø INPUT A$
1ØØ IF A$=B$ THEN GOTO 16Ø
11Ø LET G=G+1
12Ø PRINT "WRONG"
13Ø FOR J=1 TO 2ØØ
14Ø NEXT J
15Ø GOTO 9Ø
16Ø IF G=1 THEN PRINT "YOU WERE
   RIGHT IN 1 GUESS": STOP
17Ø PRINT "YOU WERE RIGHT IN□";G;
   "□GUESSES"
18Ø STOP
```

```
1Ø G=1: ON RND(6) GOTO 2Ø,3Ø,4Ø,5Ø
   6Ø,7Ø
2Ø B$="APPLE": GOTO 8Ø
3Ø B$="ORANGE": GOTO 8Ø
4Ø B$="BANANA": GOTO 8Ø
5Ø B$="LEMON": GOTO 8Ø
6Ø B$="PASSION FRUIT": GOTO 8Ø
7Ø B$="PINEAPPLE"
```



```
80 CLS: PRINT"I AM A FRUIT, WHAT
   FRUIT AM I?"
90 INPUT A$
100 IF A$ = B$ THEN GOTO 160
110 G = G + 1
120 PRINT"WRONG!"
130 FOR J = 1 TO 1000
140 NEXT J
150 GOTO 90
160 IF G = 1 THEN PRINT"YOU WERE
   RIGHT IN 1 GUESS" ELSE PRINT"YOU
   WERE RIGHT IN";G;"GUESSES"
```

```
10 G = 1:ON INT(RND(1)*6) + 1 GOTO 20,
   30,40,50,60,70
20 B$ = "APPLE":GOTO 80
30 B$ = "ORANGE":GOTO 80
40 B$ = "BANANA":GOTO 80
50 B$ = "LEMON":GOTO 80
60 B$ = "PASSION FRUIT":GOTO 80
70 B$ = "PINEAPPLE"
80 PRINT "▣I AM A FRUIT, WHAT
   FRUIT AM I?"
90 INPUT A$
100 IF A$ = B$ THEN GOTO 155
110 G = G + 1
120 PRINT "WRONG!"
130 FOR J = 1 TO 2000
140 NEXT J
150 GOTO 90
155 IF G = 1 THEN PRINT "YOU WERE
   RIGHT IN 1 GUESS":END
160 PRINT "YOU WERE RIGHT IN";G;
   "GUESSES"
170 END
```

Here Line 10 sets the guess counter at 1, then throws an electronic dice to pick a fruit. The options are stored in Lines 20 to 70. Whichever Line the computer goes to, it stores the name of the fruit as a string in B$, then goes on to Line 80. You then have to guess which fruit it has picked and INPUT your guess as A$. This is compared to B$ and if they are the same in all respects the computer prints out "YOU WERE RIGHT IN 1 GUESS" or "YOU WERE RIGHT IN" however many "GUESSES" you've had. If A$ is not equal to B$ the computer prints "WRONG", and you are given another guess. The program keeps going until your guess is correct.

Notice that even if you guess the right fruit, the condition A$ = B$ is not fulfilled if you spell your guess incorrectly. To be correct, *everything* in the two strings must be identical—the letters, spaces and punctuation (and any numerals).

One use of this technique for comparing strings is to check input from the keyboard, with a line like:

```
IF A$ = "YES" THEN PRINT "ARE YOU SURE?"
```

Note that the condition is not fulfilled if the word "yes" is typed in in small letters.

SORTING STRINGS

Strings can also be compared using the inequality signs < and >. They would be used in a Line like this:

```
IF A$ < B$ THEN PRINT "THE FIRST
   IS□";A$
```

Here the condition A$ < B$ asks whether the string contained in A$ comes before the string in B$ when they are put in alphabetical order. But watch out! The computer puts things in alphabetical order by looking at the ASCII code of each letter in turn—A has an ASCII code of 65, and Z is 90. The problem is that small letters have ASCII codes too—a is 97, z is 122. So all the strings that begin with capital letters are put first.

Worse, numerals, punctuation marks, spaces and other signs also have ASCII codes so the ranking of strings could be thrown all over the place. Still, with care, '<' and '>' can be used to sort strings into alphabetical order. In fact an alphabetic sort routine, called a 'bubble-sort' is given in the Structuring a Program article on page 216.

SLICING STRINGS

It is possible to pick out a character or sequence of characters from inside a string. On the Dragon, Tandy, Vic, Commodore and Acorn this is done using the functions LEFT$, RIGHT$ and MID$. The Spectrum uses a different technique, explained below.

LEFT$(A$,number) starts at the beginning, or left-hand end of string A$ and gives you the number of characters you specify. If A$ is "MR JOHN SMITH" and you specify two characters—LEFT$(A$,2)—the result is "MR".

Similarly, RIGHT$ counts from the other end of the string—the right-hand end. So RIGHT$(A$,5) will produce "SMITH".

With MID$ you can specify two numbers, the starting position and the number of characters to slice off. For example, MID$(A$,4,6) will start at the fourth character J and pick out 6 characters, giving "JOHN S". If you only specify one number, such as MID$(A$,4), then you'll get all characters from 4 onwards.

On the Spectrum the method of slicing strings is simpler. There is only one function A$(number TO number). A$ identifies the string to be sliced and the two numbers are the beginning and end of the slice.

With the same string A$ = "MR JOHN SMITH", A$(1 TO 2) gives you "MR", A$(4 TO 7) gives "JOHN" and A$(9 TO 13) gives "SMITH". You don't have to specify both numbers. If you miss out the first one then the Spectrum assumes you are starting at the beginning. And if you miss out the last one it assumes you want to carry on to the end.

Here's a program that uses LEFT$, MID$, and LEN (see below). It's an anagram game for two people. One person inputs a word which the computer then scrambles and prints out as an anagram for the second person to solve.

```
10 CLS
20 PRINT@65, "ANAGRAM PROGRAM"
30 PRINT@161,"ENTER WORD TO BE
   SCRAMBLED ?"
40 A$ = INKEY$:IF A$ = "" THEN 40
43 IF A$ = CHR$(13) THEN 55
46 IF A$ < "□" THEN 40
49 W$ = W$ + A$:GOTO 40
55 WORD$ = W$
70 CLS
80 FOR N = LEN(W$) TO 1 STEP -1
90 M = RND(N)
100 A$ = A$ + MID$(W$,M,1)
110 W$ = LEFT$(W$,M - 1) + MID$
   (W$,M + 1)
120 NEXT N
```





Microtip

Put your strings in order

When the computer magically picks out the addresses from hundreds of separate entries and displays them on the screen, it's easy to imagine that the machine is more intelligent than it actually is. It's almost as if the computer is actually reading the entries and then deciding what to do.

Don't be fooled. All the computer is doing is picking out a certain section of the string as it has been instructed. If that part of one of the strings happens to contain rubbish, then that is what you will get.

To ensure that you get the same information out of every string, you must ensure that the information is put into it in exactly the same place each time. Professional researchers often make use of standard-format entry cards, and they are one of the most useful aids you can make for yourself. All you need is a series of boxes to contain the characters, labelled for where each piece of information must start and finish. An example of such a card is shown on page 2Ø1, but obviously will depend on the information you are collecting.

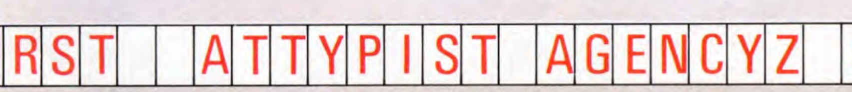
```
130 PRINT@65,"THE ANAGRAM IS□"A$
140 PRINT@129,"WHAT DO YOU THINK
    THE WORD IS ?"
160 INPUT GUESS$
170 G=G+1
180 IF GUESS$<>WORD$ THEN PRINT
    "□WRONG, TRY AGAIN":GOTO 160
190 PRINT:PRINT"□WELL DONE"
200 IF G=1 THEN PRINT"□YOU TOOK
    1 TRY" ELSE PRINT"□YOU TOOK";G;
    "TRIES"
210 PRINT@480,"□DO YOU WANT
    ANOTHER GO (Y/N)?"
220 A$=INKEY$: IF A$<>"Y" AND
    A$<>"N" THEN 220
230 IF A$="Y" THEN RUN
240 END
```

```
10 CLS
20 PRINT '"ANAGRAM PROGRAM"
30 PRINT '"ENTER WORD TO BE
   SCRAMBLED"
40 VDU 21
50 INPUT W$
55 WORD$=W$
60 VDU6
70 CLS
80 FOR N=LEN(W$) TO 1 STEP −1
90 M=RND(N)
100 A$=A$+MID$(W$,M,1)
110 W$=LEFT$(W$,M−1)+MID$
    (W$, M+1)
120 NEXT N
130 PRINT '"THE ANAGRAM IS□";A$
140 PRINT '"WHAT DO YOU THINK THE
    WORD IS?"
160 INPUT GUESS$
170 G=G+1
180 IF GUESS$<>WORD$PRINT "WRONG,
    TRY AGAIN":GOTO 160
190 PRINT '"WELL DONE"
200 IF G=1 PRINT "YOU TOOK 1 TRY"
    ELSE PRINT "YOU TOOK□";G;"□TRIES"
210 PRINT ''"DO YOU WANT ANOTHER
    GO (Y/N)?"
220 A$=GET$:IF A$<>"Y" AND
    A$<>"N" THEN 220
230 IF A$="Y" THEN RUN
240 END
```

For the ZX81, you'll need to change all variables to upper case, delete Lines 4Ø and 6Ø, leave out the apostrophes (') after PRINT statements and split multistatement lines:

```
10 CLS : LET a$="": LET g=0
20 PRINT '"ANAGRAM PROGRAM"
30 PRINT ''"ENTER WORD TO BE
   SCRAMBLED"
40 POKE 23609,20: POKE 23658,8:
   POKE 23624,63
50 INPUT w$
55 LET s$=w$
60 POKE 23624,56
70 CLS
80 FOR n=LEN w$ TO 1 STEP −1
90 LET m=INT (RND*n)+1
100 LET a$=a$+w$(m)
110 LET w$=w$(TO m−1)+
    w$(m+1 TO )
120 NEXT n
130 PRINT '"THE ANAGRAM IS
    □";a$
140 PRINT '"WHAT DO YOU
    THINK THE WORD IS?"
160 INPUT LINE g$
170 LET g=g+1
180 IF g$<>s$ THEN PRINT
    "WRONG, TRY AGAIN": GOTO 160
190 PRINT '"WELL DONE"
195 IF g=1 THEN PRINT "YOU TOOK ONE
    TRY!": GOTO 210
200 PRINT "YOU TOOK□";g;"□TRIES"
210 PRINT '"DO YOU WANT ANOTHER
    GO (Y/N)?"
220 LET a$=INKEY$: IF a$<>"Y" AND
    a$<>"N" THEN GOTO 220
230 IF a$="Y" THEN RUN
```

```
10 PRINT "♥"
20 PRINT "ANAGRAM PROGRAM"
30 PRINT "ENTER WORD TO BE
   SCRAMBLED"
50 INPUT ">■";W$
55 WO$=W$
70 PRINT "♥◆"
80 FOR N=LEN(W$) TO 1 STEP −1
90 M=INT(RND(1)*N)+1
100 A$=A$+MID$(W$,M,1)
110 W$=LEFT$(W$,M−1)+RIGHT$
    (W$,LEN(W$)−M)
120 NEXT N
130 PRINT "THE ANAGRAM IS□";A$
140 PRINT "WHAT DO YOU THINK
    THE WORD IS?"
160 INPUT GUESS$
170 G=G+1
180 IF GUESS$<>WO$ THEN PRINT
    "WRONG, TRY AGAIN":GOTO 160
190 PRINT "WELL DONE"
195 IF G=1 THEN PRINT "YOU TOOK
    1 TRY": GOTO 210
200 PRINT "YOU TOOK";G;"TRIES"
210 PRINT "DO YOU WANT ANOTHER
    GO. (Y/N)?"
220 GET A$:IF A$<>"Y" AND A$<>
    "N" THEN 220
230 IF A$="Y" THEN RUN
```





TROUBLESHOOTER

• Because of a fault on BBC micros with operating system 1.0, INSTR doesn't work exactly as it should. The problem arises if the string you are searching for is longer than the first string. You won't get an error message but in some circumstances it may cause your program to crash.

It's the sort of thing that can easily happen by mistake. Say a program asks someone to input a compass direction which is then stored in A$. If you use INSTR("NESW",A$) to check their input and they type "NORTH" in full then you're in trouble. The way round this is to check the length of the input using the LEN function before searching for it with INSTR and then ensure that you reject entries that are too long.

Change Lines 50 and 70 of the last program for the Commodore 64 to:

```
50 INPUT "> ■";W$
70 PRINT "■■"
```

When you RUN this game you'll see that the word you type in first doesn't appear on the screen. This is so your opponent cannot see what it is. Each computer uses a different method to do this. The Acorn uses VDU 21 to turn off output to the screen while you type in the word, and then it uses VDU 6 to turn the output back on again. The Commodore, Vic and Spectrum computers print the word in the background colour so that it's invisible, and the Dragon and Tandy build up the word using INKEY$ which grabs a character one at a time without printing it on the screen.

The scrambling routine is in Lines 80 to 120. What happens is that the characters are picked out at random from the word then added one at a time to A$ which gradually builds up into the anagram.

Line 90 chooses a random number M which is between 1 and the length of the word, then Line 100 picks out the Mth letter and adds it to A$. Line 110 removes this character from the original word by taking the lefthand part of the word up to M and adding on the rest of the word that comes after M. The word is now one character shorter, but the next time round the loop the variable N is also one less so the random number M is again restricted to the length of the word.

Eventually, all the characters are picked out so the anagram is printed on the screen and your opponent is asked to guess what the original word was. When they INPUT the correct word they are told how many goes they've had and you are offered another go.

It would be very easy to alter this program so that the words are read in from a DATA list rather than INPUT separately each time. You could also work out a scoring system so you give, say, ten points for guessing right first time, nine points for guessing right after two goes and so on.

Another use of this slicing method is to manipulate dates. Even when dates are keyed in figures—such as 27/03/51, meaning the 27th of March 1951—they form a string. After all you can't manipulate a date expressed that way by the normal laws of mathematics. But you may want to work with different parts of a date using arithmetic. You can, for example, work out how old someone is on a particular day if you are given their date of birth, or you can work out how many days have elapsed between two particular dates—so long as you handle the year part, the month part and the day part separately.

LEFT$, RIGHT$ and MID$—or Spectrum's equivalent method—easily separate out the day, month and year parts of a date. Then if you use the VAL functions (see later) you will turn the resulting string slice into a number which you can add, subtract, multiply or divide.

STRING LENGTHS

It is sometimes useful to know the length of a string. If, for example, you have only a limited amount of memory set aside for a piece of information typed in on the keyboard, or only a limited space on the screen to display it, it may be helpful to check the length of the entry before proceeding with the program

LEN(A$) gives the number of characters in the string A$. It is a numeric function, not a string, and can be manipulated according to the laws of algebra. For example, if A$ = "Mr John Smith", LEN (A$) = 13. But say you have been asked to enter a name into a file and there is only enough space in the tabulated format of the file's display for 11 letters you could ask the user to cut down his entry in the following way:

```
10 PRINT "ENTER SUBJECT NAME"
20 INPUT A$
30 IF LEN(A$)>11 THEN PRINT "SORRY,
   ONLY 11 CHARACTERS AVAILABLE":
   GOTO 10
```

The user could then pare down his entry and just key in 'JOHN SMITH' instead.

In other cases, it might be easier to truncate an entry automatically. This is done with a line like the following:

```
IF LEN(A$)>15 THEN LET A$=LEFT$(A$,15)
```

or on the Spectrum

```
IF LEN(A$)>15 THEN LET A$=A$(TO 15)
```

STRINGS TO NUMBERS

One use of string variables is to get idiot-proof input from the keyboard. For example, if you write a program which includes the lines:

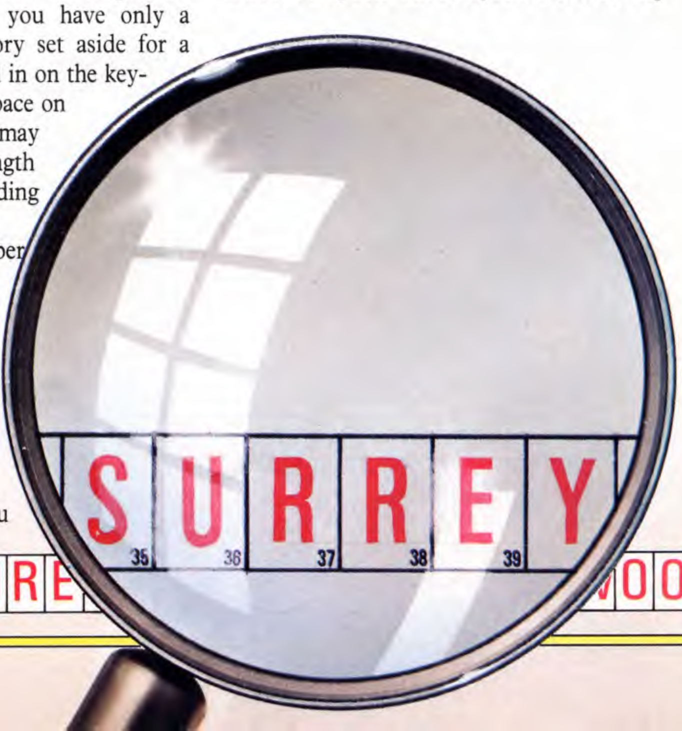
```
100 PRINT "ENTER A NUMBER"
110 INPUT A
```

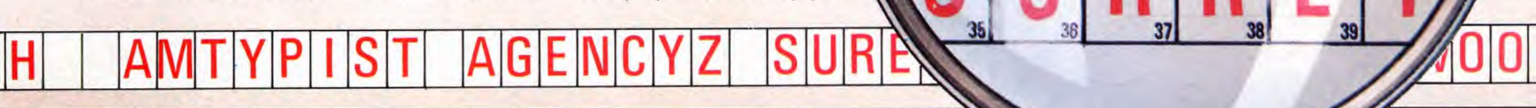
then the computer waits for you to key in a number. If by mistake you key in a non-numeric character, then most computers, except the Acorn, will break the screen display and print out a standard error message. This will tell you that something is wrong, but it won't tell you what.

But if you used:

```
110 INPUT A$
```

instead, the user can type in almost anything and the computer will accept it. The next step





is to convert the string into a number. To do this you use VAL(A$).

Unfortunately this will not work on the Spectrum. If you try to work out the value of a string of characters you'll get an error message. You can only use VAL on the Spectrum if the string consists entirely of numbers. So VAL("1984") gives 1984 which is correct, but VAL ("26/10/84") gives 0.03095 ... because the Spectrum uses VAL to evaluate the expression 26÷10÷84. Spectrum users should therefore skip the rest of this section and go straight to 'Numbers to strings'.

The VAL function on the Dragon, Tandy, Acorn, Vic and Commodore computers extracts the numerical part of the string. So if A$ is a numeral then VAL(A$) will be a number that can be used in the rest of the program. If A$ is not a numeral then VAL(A$) returns the value 0. You can then write a little subroutine that explains in detail what the user has done wrong and gives him or her another go without coming out of the program and destroying the screen display.

One important point to note about VAL is that it only extracts numbers at the beginning of the string. So VAL("25 JULY") equals 25, but VAL ("JULY 25") equals 0. So beware!

The VAL function is useful for sorting numbers out of a string in several different ways. If for some reason you have the names and marks of a class of pupils as strings: A$="32 COLIN" B$="45 MARK" C$="41 JENNY" and so on, and then want a class average, you can extract individual marks using VAL. VAL(A$) gives 32, VAL(B$) gives 45 and VAL(C$) gives 41.

In the same way, VAL can be used to ignore the units that may have been entered with such things as weights and measure. The next program shows you the difference:

```
100 LET A$="32 KG"
110 LET B$="110 KG"
120 PRINT "A$+B$=";A$+B$
130 PRINT "VAL(A$)+VAL(B$)=";
   VAL(A$)+VAL(B$)
140 END
```

NUMBERS TO STRINGS

The function STR$ does almost the exact opposite of VAL. It changes a number into a string. The advantage of this is that strings can be manipulated in different ways to numbers—by slicing and concatenating for example—so STR$ has a number of applications.

The next program converts an ordinary decimal number into binary. Although computers do all their arithmetic in binary numbers inside the machine, the BASIC language can only handle decimal or, in some cases, hexadecimal numbers. So when a binary number appears in a BASIC program it has to be handled as a string:

```
10 PRINT "DECIMAL TO BINARY"
20 PRINT "ENTER A DECIMAL INTEGER"
30 INPUT D
40 LET B$=""
50 LET B$=STR$(D-INT(D/2)*2)+B$
60 LET D=INT(D/2)
70 IF D<>0 THEN GOTO 50
80 PRINT "THE BINARY NUMBER IS□";B$
```

When you input a positive decimal number, Line 40 sets B$ equal to an empty or null string, which is then progressively filled up with digits as the computation is carried out by the loop in Lines 50 and 80.

Line 50 is the line that actually builds up the binary number. It subtracts twice the integer value of half the decimal number, from the decimal number itself. This is simply a way of testing whether the number is odd or even. If it is odd the result is 1, and if it is even then the result is 0. These, of course, are the binary digits.

On the Acorn computers this part can be done more elegantly using the MOD function as in:

```
50 B$=STR$(D MOD 2)+B$
```

The binary digits are turned into a string using STR$ and the binary number is built up by concatenating each new digit with the rest of the string.

STRING$ AND INSTR

The Acorn, Dragon and Tandy computers have two string functions more than the others.

STRING$(N,A$) produces a string of the character or characters defined by A$, N times. N must be a number or numeric variable while A$ must be a string variable, a character or a string of characters between double quotes. PRINT STRING$(6,"*") prints out ******.

On the Acorn computers, if A$ is "X−X", say, and N is 3 then STRING$(3,A$) gives X−XX−XX−X. This function is used to give repetitive patterns for decorative lines and borders and can be used whenever a long string can be generated from a sequence of shorter ones.

On the Dragon and Tandy, though, STRING$(3,A$) will give you XXX. It only repeats the *first* character of the string the rest is ignored.

Here are two programs (one for the Dragon and Tandy and one for the Acorn computers) which use STRING$ to print out a decorative border round the screen. You can use it to brighten up the title page of a game:

```
10 CLS0
20 A$=CHR$(158)+STRING$(30,CHR$
   (156))+CHR$(157)
30 B$=CHR$(154)+CHR$(174)+STRING$
   (28,CHR$(172))+CHR$(173)+
   CHR$(149)
40 C$=CHR$(154)+CHR$(171)+STRING$
   (28,CHR$(163))+CHR$(167)+
   CHR$(149)
50 D$=CHR$(155)+STRING$(30,CHR$
   (147))+CHR$(151)
60 F$=CHR$(154)+CHR$(170)+
   STRING$(28,"□")+CHR$(165)+
   CHR$(149)
70 PRINTA$;
80 PRINTB$;
90 FOR K=1 TO 11
100 PRINTF$;
110 NEXT K
120 PRINTC$;
130 PRINTD$;
140 PRINT@233,"HELLO THERE !";
150 GOTO 150
```



```
1Ø MODE1
2Ø VDU23;82Ø2;Ø;Ø;Ø;
3Ø VDU19,Ø,4,Ø,Ø,Ø,19,2,2,Ø,Ø,Ø
4Ø COLOUR2
5Ø PRINTTAB(4,4)STRING$(8,"xoox")
6Ø FOR T=1 TO 11
7Ø PRINTTAB(4)"o"TAB(35)"o"
8Ø PRINTTAB(4)"x"TAB(35)"x"
9Ø NEXT
1ØØ PRINTTAB(4)STRING$(8,"xoox")
11Ø COLOUR1
12Ø FOR T=1 TO 11 STEP 2
13Ø PRINTTAB(9,9+T)"o"TAB(21)"o"
14Ø PRINTTAB(9,1Ø+T)"x"TAB(21)"x"
15Ø NEXT
16Ø PRINTTAB(1Ø,1Ø)STRING$(5,"xoox")
17Ø PRINTTAB(1Ø,21)STRING$(5,"xoox")
18Ø COLOUR3
19Ø PRINTTAB(15,15)"HI THERE !"
2ØØ GOTO 2ØØ
```

The Dragon and Tandy program uses several graphics characters to print the border. The blocks are printed in two different colour schemes—yellow/black and blue/black. To make a symmetrical border the colours are reversed from top to bottom and side to side. Five strings are used, A$ and B$ make the top of the border, F$ does the sides and C$ and D$ make the bottom section. Line 15Ø simply prevents the OK prompt breaking up the screen display.

The Acorn program uses ordinary keyboard characters "XOOX" to make its border. STRING$ is used to make the top and bottom section but the side borders are printed much more easily with TAB. Line 2ØØ prevents the prompt reappearing and breaking up the picture.

SEARCHING FOR A STRING

INSTR is a search function. It looks through a long string for a shorter one so you can use it to find a word in a sentence, say, or a letter in a word.

You specify INSTR(A$,B$) and the computer searches A$ for the string B$ and returns the position down the string where it made its first appearance. PRINT INSTR ("HELLO","L") will display 3. If the computer cannot find the string it has been told to look for it will return Ø as shown be this next program:

```
1Ø A$="HELLO"
2Ø B$="W"
3Ø PRINT INSTR(A$,B$)
```

INSTR works slightly differently on the Acorn from the way it works on the Dragon and Tandy. Although the computers allow you to include a number in the brackets along with the two strings, the number appears *first* on the Dragon and Tandy—INSTR(P,A$,B$)—and *last* on the Acorn—INSTR(A$,B$,P). In both cases, P specifies the position down A$ that you want to start looking. Sometimes it is useful to search further down the string for a character or set of characters rather than always starting at the beginning.

Say you are playing a game of hangman and you've found the first T in NOTWITHSTANDING at position 3, you find the next one by putting P=3+1—that is, 4—in INSTR(P,"NOTWITHSTANDING","T") or INSTR("NOTWITHSTANDING","T",P). And when you've found the second one at position 6, you put P=7 for the last one. With either computer, if you omit P it assumes it is 1 and starts searching at the beginning.

If you search a string with a null string—"", both computers will give the answer 1.

Another use for INSTR is in checking input from the keyboard. Say you have to choose one option from a menu by typing in its initial letter. The options might be:

```
1Ø PRINT "(P)rint text"
2Ø PRINT "(S)ave text"
3Ø PRINT "(L)oad new text"
4Ø PRINT "(E)dit text"
5Ø PRINT "PLEASE SELECT OPTION"
6Ø INPUT A$
```

A handy way of checking whether the letter entered is valid is to add this line:

```
7Ø IF INSTR("PSLE",A$)=Ø THEN
   GOTO 5Ø
```

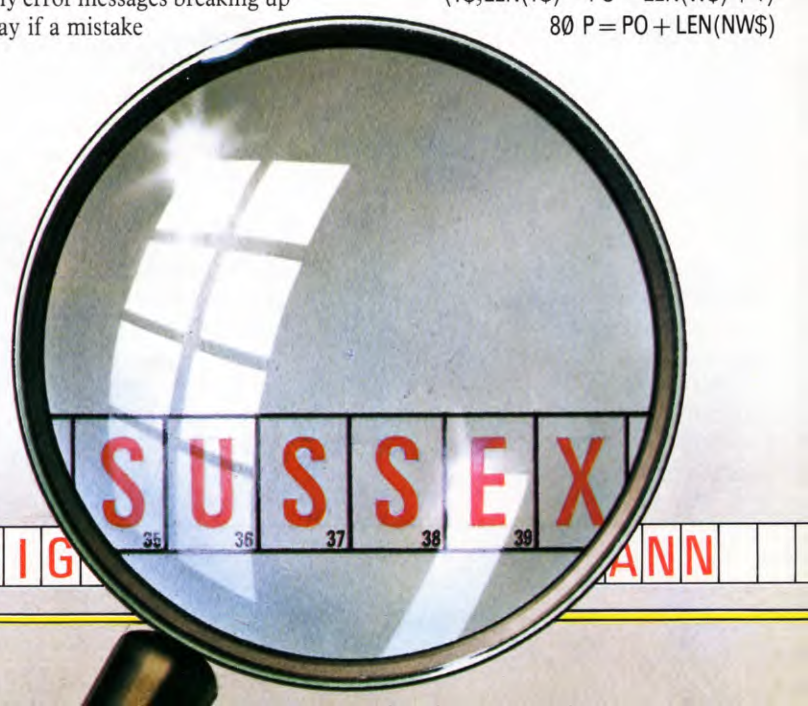
This prevents any error messages breaking up the screen display if a mistake is made.

WORD PROCESSING

The string functions have a whole range of uses, especially in word processing. A common requirement is to replace one word with another throughout the whole document (perhaps when you discover a spelling mistake). INSTR will quickly pick out every occurrence of the word and you can easily substitute the new word. Other methods will work if your computer doesn't have INSTR, but they are much slower. Of course, if the new word is a different length then you must move the rest of the text to leave just the right amount of room. The next program shows how to do it:

```
1Ø INPUTLINE "ENTER TEXT",T$
2Ø INPUTLINE "WORD TO BE REPLACED",W$
3Ø INPUTLINE "NEW WORD",NW$
4Ø P=1
5Ø pos=INSTR(T$,W$,P)
6Ø IF pos=Ø THEN GOTO 1ØØ
7Ø T$=LEFT$(T$,pos-1)+NW$+RIGHT$
   (T$,LEN(T$)-pos-LEN(W$)+1)
8Ø P=pos+LEN(NW$)
9Ø GOTO 5Ø
1ØØ PRINT T$
11Ø GOTO 2Ø
```

```
1Ø LINEINPUT "ENTER TEXT ?";T$
2Ø LINEINPUT "WORD TO BE REPLACED ?";
   W$
3Ø LINEINPUT "NEW WORD ?";NW$
4Ø P=1
5Ø PO=INSTR(P,T$,W$)
6Ø IF PO=Ø THEN GOTO 1ØØ
7Ø T$=LEFT$(T$,PO-1)+NW$+RIGHT$
   (T$,LEN(T$)-PO-LEN(W$)+1)
8Ø P=PO+LEN(NW$)
```



```
90 GOTO 50
100 PRINT T$
110 GOTO 20
```

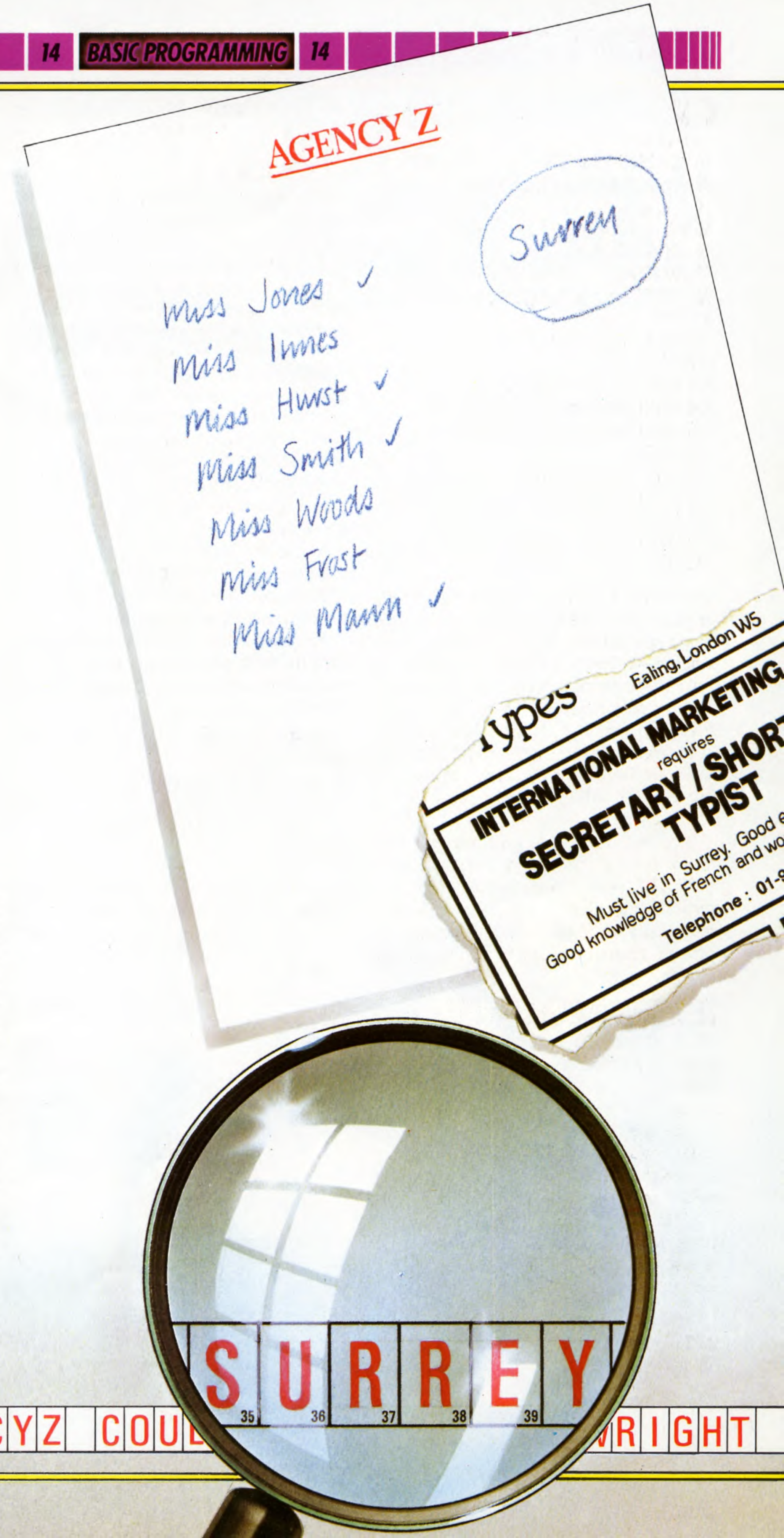
```
10 INPUT "ENTER TEXT";T$
20 INPUT "WORD TO BE REPLACED";W$
30 INPUT "NEW WORD";NW$
35 P=0
40 P=P+1
50 A$=MID$(T$,P,LEN(W$))
60 IF A$<>W$ THEN 90
70 T$=LEFT$(T$,P-1)+NW$+RIGHT$
   (T$,LEN(T$)-P-LEN(W$)+1)
80 P=P+LEN(NW$)-1
90 IF P<LEN(T$) THEN GOTO 40
100 PRINT T$
110 GOTO 20
```

```
10 INPUT "ENTER TEXT□"; LINE t$:
   PRINT t$
20 INPUT "WORD TO BE REPLACED□";
   LINE w$: LET w=LEN w$
30 INPUT "NEW WORD□"; LINE n$:
   LET n=LEN n$
35 LET p=0
40 LET p=p+1
50 IF p+w-1>LEN t$ THEN GOTO 100
60 IF t$(p TO p+w-1)<>w$ THEN
   GOTO 40
70 LET t$=t$( TO p-1)+n$+t$(p+w
   TO ): GOTO 40
100 PRINT t$
110 GOTO 20
```

Start by inputting a fairly simple sentence then try the effect of substituting some of the letters or words. With short words like 'to' or 'an', it's best to type them in with a space before and after the word, or every occurrence of 'to' and 'an' inside other words—like 'toad' and 'banana'—will also be changed. On the Commodores, enter in quotes—"an".

Here's how it works. Line 40 sets P to start the search process at the beginning of the text. Lines 50 and 60 find the first occurrence of the word or letter you want replaced, then Line 70 substitutes in the new word. It simply takes the original text up as far as the old word, adds on the new word then adds on the remaining text. The process is repeated until all occurrences of the word have been replaced, and is then printed by Line 100.

This is a very simple example—real word processors are much more complicated—but it gives you an idea of some practical applications of the string functions.



MEMORIES ARE MADE OF THIS

The computer's memory has to keep track of everything that's going on inside the machine as well as storing your programs. But how does it work and how does it all fit in?

When you enter the world of machine code programming you are approaching your home computer on its most fundamental level. Not only do you have to understand something about the hexadecimal numbers you key in—and the binary numbers the computer converts them into (see page 156)—you also have to understand how the computer itself works.

This doesn't mean that you have to understand how the chips are etched or how they are wired up. But you do have to understand a little of the overall architecture of the machine—how the major parts and systems are related and what each one does.

WHAT MEMORIES ARE MADE OF

Home computers store their memory on a series of silicon chips. Each memory chip carries thousands of tiny circuits which can be either switched on, or switched off. Each of these circuits represents a single digit in binary—a bit. When the circuit is switched on it represents 1, when it is switched off it represents Ø.

Within the chip, these tiny circuits are organized in groups of eight. Each group represents a byte—that is an eight-bit binary number, or two digits in hex.

But there would be no point in storing numbers in these circuits without being able to find them again. So each eight-bit memory location is given an address. In a machine with a total memory space of 64K, for example, you need 64K addresses—one for each location.

HOW MUCH MEMORY?

A 'K' in computing is roughly analogous to the 'k' which stands for kilo—a thousand—in the metric system. But 1,000 is not very convenient when converted into binary or hex.

The nearest convenient hex number is 4ØØ, which is 1,024 in decimal and represented by 1 with ten zeros after it in binary. This number is defined as 1K.

So if you need to identify 64K memory locations, you could number them from 1 to 65,536 in decimal. But as these numbers themselves will be used by the computer it is much more convenient to number them in hex from ØØØØ to FFFF. This uses every possible four digit—or two byte—hex number. The four-digit hex number allocated to a memory location is known as its address. And each location can be addressed—that is written into or read out of—by quoting its unique four-digit hex number.

Of the computers covered here, only one claims a 64K memory—the Commodore. But the 32K BBC Micro, the 32K Electron, the 48K Spectrum, the 32K Tandy and the 32K Dragon all have 64K of memory space in all. The 'K' figure refers to the amount of memory space you can *use*—the other 32K or 16K is reserved for the machine's use. The Commodore's memory is rather more flexible and the 64K claim is based on the fact that it is possible for the user to intrude onto the machine's own memory space.

In each of the above machines, the makers number their memory locations from ØØØØ to FFFF. The smaller 16K Spectrum—which has 32K of memory in all—numbers its memory locations from ØØØØ to 7FFF.



- WHAT THE MEMORY IS MADE OF
- HOW MUCH MEMORY DOES YOUR COMPUTER HAVE?
- ROM AND RAM
- HOW THE MEMORY IS DIVIDED UP
- WHAT'S IN THE MEMORY
- WHERE BASIC PROGRAMS ARE STORED
- HOW THE COMPUTER STORES LARGE NUMBERS

TURNING THE PAGE

Memory space is further organized into pages. (These are not to be confused with the larger high-resolution pages used by the Dragon and Tandy.) Each page holds 1ØØ memory locations in hex—that's 256 in decimal.

The so-called 'zero page' runs from ØØØØ to ØØFF, the 'one' page runs from Ø1ØØ to Ø1FF and so on.

ROM AND RAM

There are two different types of memory—ROM and RAM. ROM stands for Read-Only Memory. That means you can read out of ROM memory locations, but you cannot write into them. The information they contain is fixed permanently when the ROM chips are manufactured and they are protected against tampering. No matter what you do to your machine—short of physical damage—the ROM will restore its functions to working order if you switch it off and then back on again.

The ROM contains the computer's operating instructions and the interpreter that translates your BASIC programs into machine code. It also imposes a structure on the rest of memory which can be seen in the memory maps below.

RAM stands for Random Access Memory. This is the memory which is open to the user, but is not quite as wide open as its name implies. The ROM takes over some of it for screen displays and other specific functions and if you try and write something in these memory locations the ROM will change it back for you. But fundamentally it is an empty slate on which you can write whatever you want, then read back at your leisure.

KEEPING TRACK

The memory maps shown below are a pictorial representation of where things are in the memory. These are not their exact physical positions, as the memory space is divided up onto a number of different chips inside the computer. But the map does show you schematically how various parts of the memory are used for different things.

Some of the fixed frontiers between sections of memory—like the one between ROM and RAM—do coincide with the change from one chip to the next. Other frontiers are flexible and their position is indicated by a *pointer* in the *system variables* area.

A pointer is a memory location—or rather a pair of memory locations—which stores the address of another location, in this case the start of a particular section of memory. The address of any byte of memory is two bytes long, so it has to be stored in two adjacent memory locations. The pointers are so called because, when the computer looks at them, it is directed to the location it should move to.

The Spectrum's 16K ROM runs from ØØØØ to 3FFF. It contains the BASIC interpreter, the editor, various input and output routines and the character set which contains all the data for the letters of the alphabet, figures and the other graphics symbols available on the Spectrum. The other 48K—from 4ØØØ to FFFF—on the 48K model, and the other 16K—from 4ØØØ to 7FFF—on the 16K, are RAM.

The RAM is divided up into areas, each of which has a specific job to do.

The *display area* controls what is shown on your TV screen. Each memory location corresponds to a line of eight pixels.

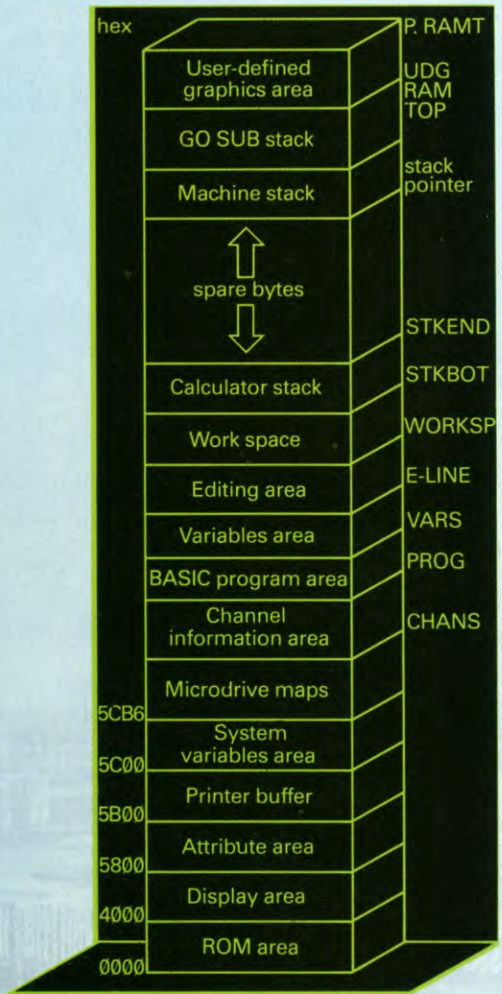
The *attributes area* controls the PAPER and INK colours of each of the screen's 768 character areas and whether the display is FLASHing or steady, BRIGHT or normal.

The *printer buffer* holds the next line of text that is going to be fed out to the printer.

The *system variables* are the locations which hold the addresses—or pointers—of the beginnings of the specific areas above them in memory.

The *microdrives map area* only exists if you have a microdrive attached to your Spectrum.





Spectrum memory map

Otherwise CHANS, whose address is stored at locations 23,631 and 23,632 (5C4F and 5C5Ø), moves down to 5BC6.

It's the *channel information area* which carries input and output data. It transmits input from the keyboard to the lower part of the TV screen and controls program output to the rest of the screen, to the workspace higher up in memory and to a printer.

The *BASIC program area* holds the current lines of any BASIC program you've keyed in, and its size depends on the length of the program. It starts at the address given by the systems variable PROG, which is held in locations 23,635 and 23,636 (5C53 and 5C84) in the system variables area. This points to 23,755 (5CCB in hex) if no microdrive is attached.

The following program looks at this area and PRINTs out the number in each memory location. Alongside that it PRINTs the ASCII symbol corresponding to the number, which shows you what you typed in. Note that the keywords are not stored character by character as ASCII strings but are encoded as a single-byte—or in some cases a double-byte number. These are known as *tokens* and are translated back to the keywords automatically by the Spectrum.

```
10 FOR n=23755 TO 23848
20 PRINT n;TAB 10;PEEK n;
30 IF PEEK n>31 THEN PRINT TAB 20;
   CHR$ PEEK n;
40 PRINT : NEXT n
```

The *variables area* stores the values of the variables being used in the current BASIC program. It starts at VARS, whose location is held in locations 23,627 and 23,628 (5C4B and 5C4C hex) of the system variables area. When a program is RUN, the start of the variables area stays where it is—nothing below it in memory needs to expand—but the variables area itself grows as new variables are defined.

The *editing area* is where the editing of BASIC lines is done. As a line can be edited while it is being keyed in, BASIC program lines always appear in the editing area first. When ENTER is pushed, they are copied into the BASIC program area. The editing area starts at the E-LINE whose address is held in locations 23,641 and 23,642 (5C59 and 5C5A in hex) in the system variables area.

The *workspace* is used for general tasks like storing input data and concatenating (running together) strings. WORKSP is stored in 23,649 and 23,65Ø (5C61 and 5C62 in hex) in the system variables area. When it is not being used the workspace collapses to nothing.

The *calculator stack* is used to hold floating point numbers, five-byte integers and five-byte sets of parameters when dealing with strings. It starts at STKBOT, whose address occupies locations 23,651 and 23,652 (5C63 and 5C64 in hex) and ends at STKEND, which is found in 23,653 and 23,654 (5C65 and 5C66 in hex).

Beyond that lies an area of spare memory.





This allows the memory areas either side of it room to grow, until STKEND meets stack pointer and the Spectrum will tell you it is out of memory.

Above the spare bytes is the *machine stack*. This is used by the machine itself when a BASIC program is RUN. But when you write machine code you get a chance to manipulate it yourself. Its workings will be explained in a later issue.

The GOSUB stack stores the number of the line the computer has to return to when it has completed the subroutine.

RAMTOP is to all intents and purposes the end of the RAM available for you to write programs in. Its address is held in 23,73Ø and 23,731 (5CB4 and 5CB5 hex) in the system variables area.

Above it are 168 memory locations which hold the representations of 21 user defined graphics. However, as RAMTOP is a system variable, it can be moved down in memory, pushing the GOSUB and machine stacks down into the spare bytes. This is done when you are writing machine code. Usually, a machine code program is tucked in above the lowered RAMTOP so that it cannot be overwritten by a BASIC program.

P-RAMT is the physical top of the RAM—in other words there are no more memory locations on the Spectrum's chips after this point. Although P-RAMT is fixed, as far as the Spectrum is concerned it is a system variable with its address stored in locations 23,732 and 23,733 (5CB5 and 5CB6 in hex). And you can POKE values into these and other locations to make a 48K Spectrum think it is a 16K machine.

To check the value of P-RAMT switch on the machine then enter the line:

```
PRINT PEEK 23732+256*PEEK 23733
```

This should return 65535 if you have a 48K Spectrum and 32767 if you have a 16K model. If it doesn't and you haven't POKEd anything into these locations then there is something wrong with your computer's memory and you should have your Spectrum looked at by a dealer.

You can look at any of the system variable pointers using the same PRINT line, by substituting the variable's two memory locations. The BASIC keyword PEEK looks at the contents of any byte of memory and returns the decimal equivalent of the number it finds there.

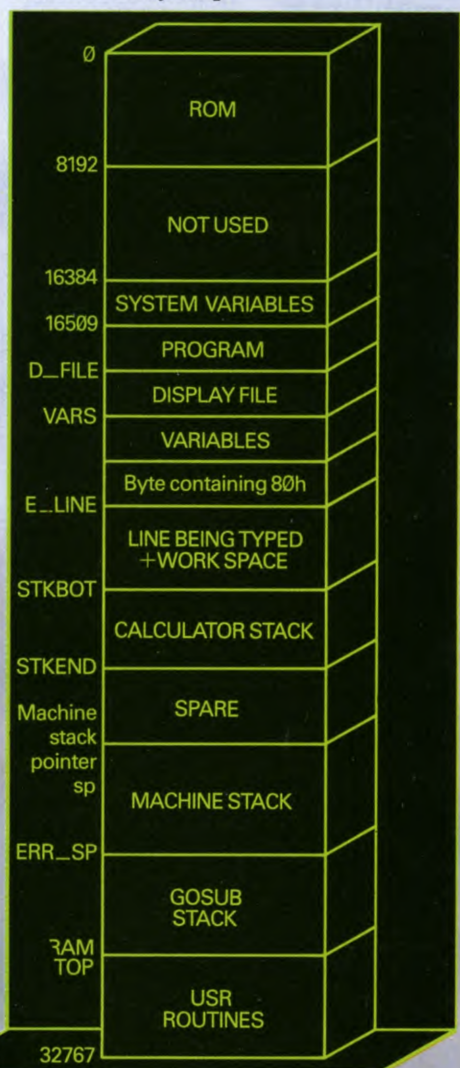
As mentioned above, the address is two bytes long and requires two memory locations to store it. The Spectrum breaks the four-digit hex address—say the normal RAMTOP address FF57—into two parts, FF and 57. The lower byte, 57, goes into the lower address and the higher byte, FF, goes into the higher address. This may look like an odd way round, but the Spectrum finds it easier to cope with this way.

That's why the second of the two memory locations in the program above is multiplied by 256—otherwise the Spectrum would return the decimal equivalent of FF, rather than FFØØ.

The ZX81 has 8K of ROM which runs from ØØØØ to 1FFF in hex, or 0 to 8,190 in decimal. The next 8K of memory—from 2ØØØ to 3FFF—are not used.

The memory map here shows the ZX81 with a 16K RAM pack. That, naturally, gives you 16K of RAM, from 4ØØØ to 7FFF in hex or 16,384 to 32,767 in decimal. If you only have an 8K or a 4K RAM pack all the different areas of memory shown are simply squeezed

ZX81 memory map



down into 8K or 4K, making the physical top of RAM 5FFF—24,575 in decimal—or 4FFF—20,479 decimal—respectively.

If you are using your ZX81 without a RAM pack, everything is compressed into 1K—from 4ØØØ to 43FF hex, 16,384 to 17,407 decimal.

In all these cases, the *system variables* occupy the area from 4ØØØ to 4Ø87 hex, 16,384 to 16,509 decimal. None of the other areas are fixed and the details of their boundaries—D-boundaries—D-FILE, VARS, E-LINE, STKBOT, STKEND, ERR-SP and RAMTOP—are stored as pointers in this system variables area.

The first of these flexible areas, from 16,509 to D-FILE, contains any BASIC program you key in. It is possible to write a BASIC program which looks at itself in the BASIC program area. The following program looks at this area and PRINTs out the number in each memory location. Alongside that it PRINTs the ASCII symbol corresponding to the number, which shows you what you have typed in. Note that keywords are not stored character by character as ASCII strings but are encoded as a single byte in the ZX81's memory. These are known as *tokens* and are translated back into the keywords by a complex routine in the ZX81's ROM. This simple program cannot do that.

```
10 FOR n=16509 TO 16704
20 PRINT n;TAB 10;PEEK n;
30 IF PEEK n>31 THEN PRINT TAB 20;
   CHR$ PEEK n
40 PRINT
50 NEXT n
```

From D-FILE to VARS, is the *display file* which controls what is shown on the TV screen.

The *variables area* expands as your BASIC program is RUN and new variables are defined. It ends with a memory location containing 8Ø in hex.

Between E-LINE and STKBOT is the *editing area* and *workspace*. The line you are currently entering goes in here so that it can be edited before you add it to the rest of the program. When you press NEW LINE the line is transferred into the program area. This area doubles as the workspace which is used for general tasks like storing input data and concatenating (running together) strings.

When calculations are being done in BASIC the numbers used are held on the *calculator stack*, which lies between STKBOT and STKEND.

From STKEND to the machine stack pointer is a spare area of memory which the areas either side of can expand into. The *machine stack* is used by the machine itself when a BASIC program is RUN and the GOSUB *stack* is used to store the line number the ZX81 should return to after it has executed a subroutine. How stacks work will be explained in a later chapter.

Normally RAMTOP is at the physical top of memory—that is 32,767, 24,575, 20,479 or 17,407 depending on the RAM pack you have. You can check this by PEEKing at the appropriate pointer in the system variables area. To do that key in the following line:

```
PRINT PEEK 16388+256*PEEK 16389
```

This should return the value of the end of memory for your machine.

It is also possible to POKE a value into these pointers and shift RAMTOP down memory, away from the physical top of memory. This leaves a protected area where you can write machine code routines and they can't be overwritten. The problem with using this area of memory for machine code routines is that they can't be SAVEd, though.

With the ZX81 the only area of memory that you can SAVE is the BASIC area. How to put your machine code programs in the BASIC area and still protect them will be covered in a later chapter.

The Commodore's memory is a different creature from those of other home computers. To start with, there is no firm distinction between ROM and RAM. Some parts of the Commodore's memory can be either ROM or RAM!

The Commodore's memory chips actually have 64K of RAM. But as the computer's microprocessor could only cope with 65,535—or FFFF in hex—addresses, there were no extra addresses to assign to the ROM chips. The way round this was to give the ROM locations the same addresses as some of the RAM. At any one time the computer knows whether it is supposed to be looking at the ROM or the RAM address by looking at the bits of memory location one. Various areas of memory are ROM when their corresponding bits are 1. But if they are set to Ø the ROM is said to be 'flipped out' to reveal the RAM beneath.

In previous chapters you will have come across the BASIC statement POKE. This is always followed by two numbers, usually in decimal. The first is less than 65,535 and is the address of a memory location. The second is less than 255, the largest decimal number then can be held in one memory location. POKE writes the second number into the memory location given by the first.

But you can write only into RAM locations—ROM means Read-Only Memory, remember. So if you POKE a number into a memory address that's shared between ROM and RAM locations, the Commodore will have to POKE it into RAM. It has no alternative.

The BASIC statement complementary to POKE is PEEK. This reads the address of any given memory location. But both ROM and RAM memory locations can be read, so the computer has to look at byte one of the memory to see whether the bit for that part of memory is Ø for RAM or 1 for ROM.

When the machine is switched on the appropriate bits are all set to 1 so that all the ROM is switched in.

The top 8,192 (2ØØØ in hex) memory locations—from EØØØ to FFFF—contain *Kernal ROM*. These are the instructions that tell the machine how to work. Even so you can flip this bit of ROM out if you want to, leaving 8K of virgin RAM. The Commodore won't operate, of course, unless you fill this section of RAM with another operating system, or copy out the one in the Kernal ROM and modify it.

The block beneath that—DØØØ to DFFF—is even more complicated. Here there is a choice between the input/output devices ROM, the character set ROM and 4K of RAM. Normally the input/output devices are switched in, but when the character set is needed, the part required is copied out into the BASIC user RAM in the area from 1ØØØ to 1FFF. The position each byte of the copy takes corresponds exactly to its position in the character set ROM.

Both the character set and the input/output ROMs can be switched out, leaving 4K of RAM, but you wouldn't be able to output anything to the screen or input anything from the keyboard.

The next 4K of memory—from CØØØ to CFFF—is pure RAM and is the area usually used for machine code programs.

From AØØØ to BFFF is the BASIC interpreter ROM, but again this can be switched out so that you can use another language or modify BASIC by copying it into the RAM.

Below that is 26K of BASIC user RAM, though some of it is used for specialized purposes. The 8K from 8ØØØ to 9FFF takes input from cartridges and the 8K from 2ØØØ to 4ØØØ maps the high resolution screen.

The screen area below that maps the text screen from Ø4ØØ to Ø7E7 and Ø7F8 to Ø7FF contains the sprite data pointers. In high resolution mode, this screen area doubles as the colour table. You can look at this BASIC user RAM with the following program.

```
10 FOR N=2048 TO 2143
20 PRINT N;TAB(10);PEEK(N);
```

```
3Ø IF PEEK(N)>31 THEN PRINT TAB(2Ø);
   CHR$(PEEK(N)):NEXT N:END
4Ø PRINT:NEXT N
```

It actually looks at itself and returns the decimal equivalents of the numbers stored in each memory location. Alongside that, it PRINTs their associated ASCII characters. So the computer will PRINT out the variables, numerical values, arithmetic signs and punctuation in the program, but it won't PRINT out the keywords. These are not stored as ASCII strings but as single-byte—or in some cases double-byte—numbers known as *tokens*.

The memory from Ø4ØØ to ØØØØ contains the *systems workspace* and *system variables*. One of them—memory location one—contains the bits which control whether the ROMs are flipped in or out.

You can look at this byte with:

```
PEEK(1)
```

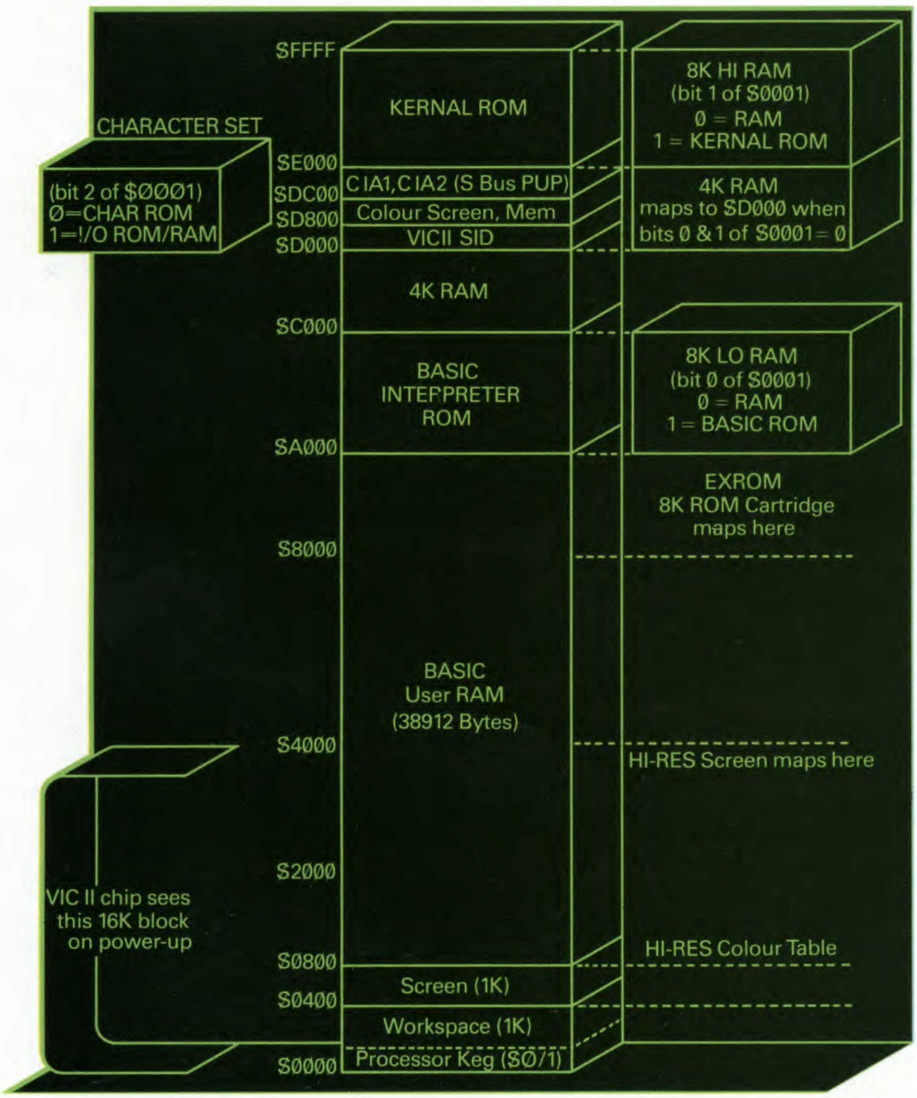
and see what is flipped in. Normally the decimal value returned is 55. And if you take the decimal value given and convert it into binary you can see what is happening in the ROM:

Bit zero—that is, the one at the right-hand end because everything is numbered from zero in computers—controls the BASIC ROM. If it is Ø the BASIC ROM is switched out. If the next bit to the left, bit one, is Ø the Kernal ROM is switched out and the character set is switched in. To switch both the input/output devices and the character set out to reveal the extra 4K of RAM, you switch out both the Kernal ROM and the BASIC ROM.

Bits three to five control a cassette machine and bits six and seven are not used.

Decimal 55 is ØØ11Ø111 in binary, so bits zero, one and two are all 1, and all the ROMS are switched in.

Commodore 64 memory map



The Vic 20's 6502 chip can access up to 64K memory locations. But the basic Vic only uses 29K of memory in all. That means that you can expand the Vic's memory by up to 35K with ROM and RAM packs plugged into the computer's memory expansion port.

The memory map given here shows where the expansion packs fit in and how the Vic's memory is structured.

The *system variables* occupy the first 1,024 memory locations, from ØØØØ to Ø3FF in hex or 0 to 1,023 decimal. This area stores the pointers which mark the various boundaries in the rest of RAM and give the Vic's memory an overall structure. The memory configuration can be changed by POKEing different values into the memory locations here.

The *user memory* extends from Ø4ØØ to 7FFF—1,024 to 32,767 decimals. That's 31K of memory in all, though much of it won't be present if you don't have add-on RAM packs. The first 7K—from Ø4ØØ to 1FFF hex, 1,024 to 8,191 decimal—is exclusively RAM, made up from a 3K expansion pack—from Ø4ØØ to ØFFF hex, 1,024 to 4,095 decimal—and 4K of the Vic's built-in user memory—from 1ØØØ to 1FFF hex, 4,096 to 8,191 decimal.

Most of this area—from ØFFF to 1DFF hex, 4,096 to 7,679 decimal—is used to store the BASIC programs you key in. It is possible to write a BASIC program which looks at itself in this area. The following program PRINTs out the number it finds in each memory location in this area. Alongside it, it PRINTs the ASCII character associated with that number.

This will give you the variable names, numbers, arithmetic signs and punctuation. But the Vic will not PRINT out the keywords because these are not stored as ASCII strings. Instead they are stored in single—or sometimes double—bytes as *tokens*.

```
1Ø FOR N=4Ø96 TO 4191
2Ø PRINT N;TAB(1Ø);PEEK(N)
3Ø IF PEEK(N)>31 THEN PRINT
   TAB(2Ø);CHR$(PEEK(N)):NEXT N
4Ø PRINT:NEXT N
```

If there is no other expansion pack present, the last 512 memory locations—from 1EØØ to 1FFF hex, 7,680 to 8,191 decimal—is *screen RAM* which controls what you see on your TV or monitor. Otherwise the screen memory is moved to start at 1ØØØ hex, 4,096 decimal.

The remaining three 8K sections that fill out the user memory can be RAM or ROM depending on the plug-in packs used and are completely free for your use, except for the ½K screen memory area.

The *character generator* occupies 8ØØØ to

8FFF hex, 32,768 to 36,863 decimal. This is 4K of ROM which contains the pixel patterns used to make up each of the 255 ASCII characters on the screen.

The *system input/output and control interfaces* occupy 9ØØØ to 912F hex, 36,864 to 37,167. This area communicates directly with the three I/O chips—two 6522 VIA chips and the 5621 VIC chip—and they can be controlled by specific POKEs in this area.

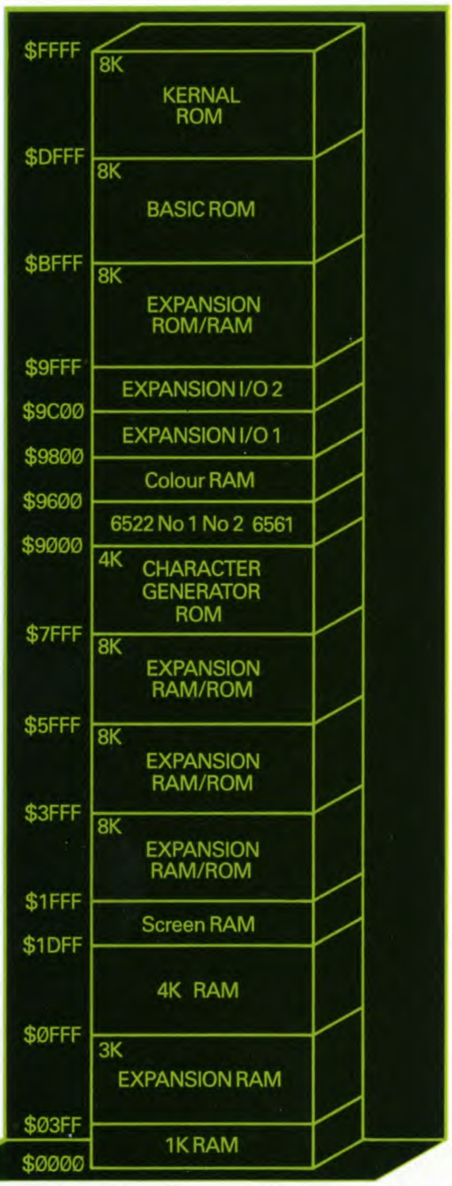
The memory locations from 94ØØ to 95FF hex, 37,888 to 38,399, deal with the *colour memory*, if the Vic has not been expanded. Each of the 506 bytes in this block fixes the foreground and background colour of the corresponding byte in the video memory. If there is more than 7K of user RAM in the system the start of the colour memory is moved up to 96ØØ hex, 38,400 decimal.

The 8K expansion from AØØØ to BFFF hex, 40,960 to 49,151 decimal, can be used by programs stored in ROM. The operating system will power up any machine code program in this area when the machine is switched on instead of BASIC.

The *BASIC interpreter* in CØØØ to DFFF hex, 49,152 to 57,343 decimal, translates your BASIC programs into machine code when they are RUN. And the Kernal ROM—from EØØØ to FFFF hex, 57,344 to 65,535 decimal—contains the *operating system* which controls how the machine works.

It is the operating system that imposes the overall organization on RAM when the computer is switched on. Otherwise none of the boundaries between areas of RAM would exist. They are only specified by this specialized part of ROM.

Vic 20 memory map



In the BBC Micro and the Electron, the ROM occupies the memory locations from &FFFF to &8ØØØ. This area contains details of the computer's operating system, the BASIC interpreter and an area which deals with input and output to peripherals such as printers, cassette recorders and the like.

The area from &7FFF to HIMEM is reserved for the *screen memory*. This expands and contracts according to the mode being used. HIMEM drops to &3ØØØ for Modes Ø, 1 and 2. It's &4ØØØ for Mode 3, &58ØØ for Modes 4 & 5, &6ØØØ for Mode 6 and &7CØØ for Mode 7 on the BBC.

The area between HIMEM and LOMEM is the *workspace* used by the BASIC program when it is RUN. LOMEM and TOP—which are pointers in the system variables area—usually point to the same spot, the first free address after the end of the BASIC program. They can be separated, though, by writing another address into the system variable location occupied by LOMEM. Once they've been split you can use the area between to house a machine code program.

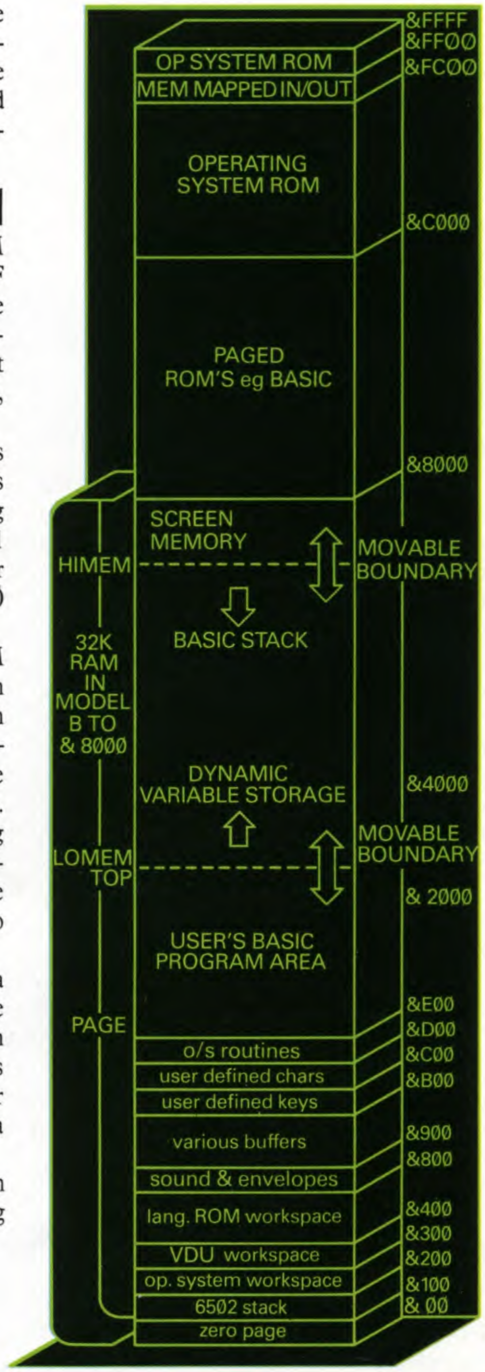
Your BASIC programs occupy the area between TOP and PAGE. And normally the system variable PAGE will be &EØØ. Again you can move it by writing another address into the location occupied by the pointer PAGE to give yourself free space to house a machine code program.

You can take a look at what's happening in the BASIC program area with the following program.

```
1Ø FOR N=PAGE TO TOP
2Ø PRINT;~N;TAB(15);?N;
3Ø IF ?N>31 THEN PRINT TAB(3Ø);
   CHR$(?N);
4Ø PRINT
5Ø NEXT
```

It PRINTs out the address you are looking at, the hex number contained in that address and the character it represents. Note that none of the keywords appear in the listing. These are not stored the way you key them in, as a string of characters, but are encoded as a number which fits into a single—or sometimes a double—byte. These numerical represent-

Acorn memory map



ations of keywords are known as *tokens*.

Page 13, between &DØØ and &DFF, can be used for short machine code programs if you are not using a disk system. If you are, the disk will overwrite this page.

Page 12—&CØØ to &CFF—is reserved for user defined graphics, and page 11—&BØØ to &BFF—stores the definitions you have given to the user-programmable keys.

Page 1Ø—&AØØ to &AFF—is the *input buffer*. That is, it takes information a block at a time from a cassette (for some cassette operations), or through an interface from another computer, before it is consigned to the correct part of memory.

Page 9—&9ØØ to &9FF—is the *output buffer*. It acts as a temporary store for information being fed out of the computer.

Page 8—&8ØØ to &8FF—contains the *sound work space*, where the various sounds the computer can make are synthesized. It also contains the *sound buffer* and the *printer buffer* which store sound and print data immediately before it is output.

Pages 7, 6, 5 and 4 are used by the BASIC interpreter in ROM to translate your programs into language the computer can understand.

Page 3—&3ØØ to &3FF—contains the *keyboard buffer*, details of the text cursor and a large number of graphics parameters.

Page 2—&2ØØ to &2FF—contains the operating parameters that change while the machine is running, and so cannot be stored with the rest of the operating system in ROM which is fixed.

Page 1—&1ØØ to &1FF—is the *machine stack*. This is a specialized area of memory that is used by the machine when BASIC programs are being RUN. But you can manipulate this area when you write machine code programs.

The *zero page*—&Ø to &FF—is used for many special functions in the BBC and Electron home computers. This is because it can be accessed quickly as its addresses only occupy one byte—the other byte, the higher one, is zero (ØØ) so any one-byte address must be on the zero page.

The Dragon and Tandy have a 32K ROM and input output area, &H8ØØØ to &HFFFF. This area controls the input to, and output from, the computer and any plug-in cartridge—plus the machine's operating instructions and the BASIC interpreter.

Between &H36ØØ and &H7FFF, BASIC programs and their variables are stored. When the program is RUN, it is performed here too. And the stack (a specialized area of memory whose function will be explained later), occupies the uppermost part of the RAM, usually nestling immediately under the ROM at &H7FFF downwards. But the stack can be moved down to make room for a machine code program. This is done by altering the value of the *system variable* which points to the base of the stack. The procedure for doing this will be explained in a later chapter.

The following program looks at itself in the BASIC program area, reads the hex values in the memory locations which store the program and PRINTs them out on the screen along with their associated ASCII characters. So the computer will PRINT out the variables, numerical values, arithmetic signs and the punctuation that appear in the program, but it won't PRINT out the keywords. These are stored as single-byte—or in some cases two-byte—numbers known as *tokens*. This simple program cannot convert these tokens back into the keywords—a special routine stored in ROM and part of the system variables is needed to do this—so it PRINTS graphics blocks instead.

```
1Ø PCLEAR4
2Ø FOR N=7681 TO 7744
3Ø PRINT N,PEEK(N);"□";CHR$(PEEK(N))
4Ø NEXT
```

Both computers have eight high resolution graphics pages—which are not the same as the memory pages mentioned on page 209. Those contained 256 locations, or $\frac{1}{4}$K. The graphic pages are $1\frac{1}{2}$K long and each contain enough graphics information to fill the screen in the lowest resolution mode—PMODEØ—whilst the highest resolution mode—PMODE4—needs four of these per screenful. The computer automatically reserves four pages, but PCLEAR will reserve more. You can, therefore, store anything from one to eight screenfuls of graphics in the memory.

When not in use, these pages are left empty. But if extra memory is required they can be cleared down to page one in BASIC. This gives access to an extra $10\frac{1}{2}$K memory for your programs.

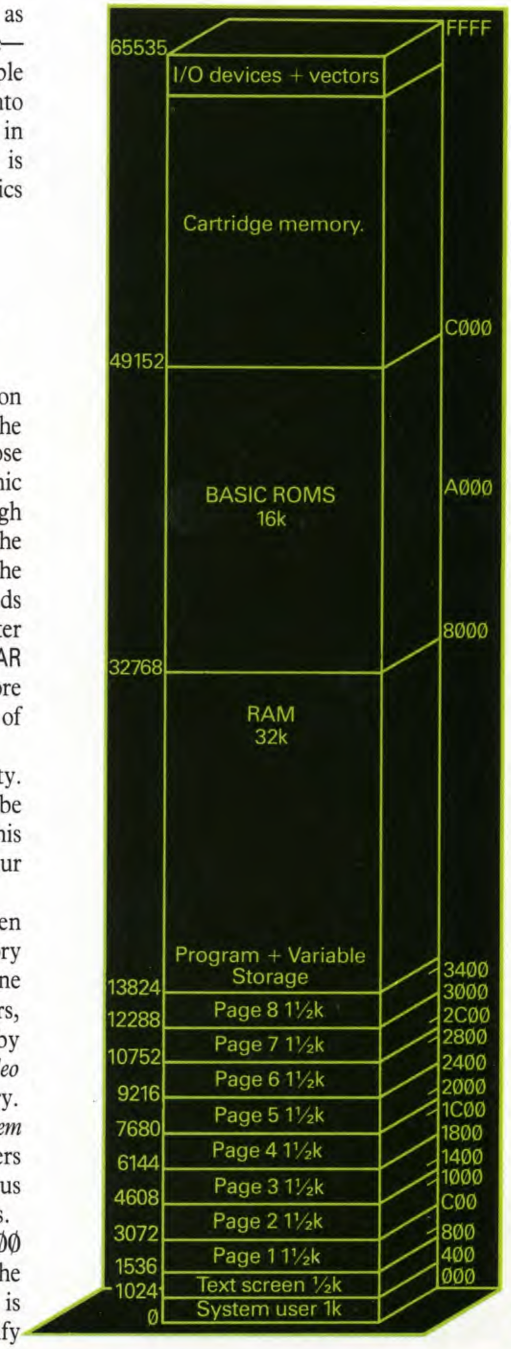
The *text screen* occupies the $\frac{1}{2}$K between &H4ØØ and &H6ØØ. This gives one memory location for each character space on the 16-line 32-character-per-line screen. The characters, which are made up of 96 pixels (eight pixels by twelve pixels), are generated in a separate *video generator chip* which is not part of the memory.

From &HØØØ to &H4ØØ are the *system variables*. These are a collection of pointers which give the addresses of the start of various areas in memory and other system variables.

The first group of 256 bytes—from &HØØ to &HFF—is known as the *direct page* on the Dragon and Tandy. But the direct page is different from the zero page as you can specify any page you want to be the direct page—that is the page whose locations can be addressed using a one-byte address rather than the normal, extended, two-byte address. This is done by setting the *direct page register* in the microprocessor. How this is done will be explained in a later chapter.

As you can see from the above there is not any spare room left for your machine code programs when the computer is left to its own devices. Again, how you make room for your machine code programs will be explained later.

Dragon and Tandy memory map





GET YOUR PROGRAMS IN SHAPE - 2



Once the general design of the program has been worked out, and the individual modules have been written you can start thinking about how you are going to test the modules. Then you can think about how you are going to fit them all together.

Most subroutines or modules need to be joined onto the rest of the program in some way. This is normally done using variables. Some variables, specified at the start, will be passed to the routine—these are the *input parameters*. Other variables will be returned to the program by the routine—these are the *output parameters*. It is very important that the variables are specified in a precise way and cannot be confused.

When you start to write a program you should make a list of all the variables you need to use, together with a description of their use and their possible values, if known. Otherwise, while you may know at the start what all the letters stand for, you are sure to forget if you return to the program later. It helps to use long variable names if your computer allows it (see page 95 for a chart of what is permissible) unless you are very short of memory space.

Here's an example of how you might specify the variables for a bubble sort routine:

Bubble Sort routine:
Sort specified portion of an array into alphabetical order.

Input variables:

A$(N) one-dimensional array to be sorted (size of N > = 1)

N1 1st item in array to be sorted (1 < = N1 < = N2)

N2 last item in array to be sorted (N1 < = N2 < = size of A)

Output variables:

A$(N) sorted array

Temporary variables:

Z, Z$, I

A note of temporary variables used within a subroutine is useful to avoid conflicts between modules or with the rest of the program. It is also useful to reserve some letters specially for temporary variables. For example Z0 to Z9 could be used. This also avoids wastage of variable space.

If you get this sorted out early on in the design of a program then you will save yourself a great deal of trouble later. Many program errors or bugs are caused by variables being *corrupted*—that is, changed in value—when you were not expecting it. But if you follow the method above you shouldn't have any trouble.

Writing down the list of variables is also useful if you ever change the program later as you will have a record of how the variables were used. This will make it quicker to alter the variables and also stops you introducing extra errors into the program by corrupting variables which are already being used for some other purpose. Remember to make a note of the changes along with the date.

Here, then, is the program for the bubble sort routine. There is an explanation of how it works and a flow chart on page 219.

```
1000 REM BUBBLE SORT (A$(N), N1, N2)
1010 LET Z=0
1020 FOR I=N1 TO N2-1
1030 IF A$(I)<=A$(I+1) THEN
   GOTO 1080
1040 LET Z$=A$(I)
1050 LET A$(I)=A$(I+1)
1060 LET A$(I+1)=Z$
```

Take your structured programming a stage further by looking at the build-up of a bubble-sort program that will sort anything from pool balls to entries in a dictionary

- KEEPING TRACK OF THE VARIABLES
- WRITING THE SUBROUTINES
- HOW TO TEST THE MODULES
- PUTTING IT ALL TOGETHER



```
1070 LET Z=1
1080 NEXT I
1090 IF Z=1 THEN GOTO 1010
1100 RETURN
```

This is a subroutine and would need to be called from the main program. For example, to sort items 5 to 20 the routine could be called like this:

```
100 LET N1=5: LET N2=20:GOSUB
   1000:REM bubble sort
```

There will also have to be a section of program that allows you to input the items to be sorted, and another section that prints out the sorted list. At this stage you should decide how you want the screen display to look, even drawing it out on some graph paper if that helps.

PROCEDURES AND FUNCTIONS

BBC BASIC has a far better method of calling and defining routines than most other versions of BASIC. It does this using PROCedures and functions that are called by name rather than by their Line numbers. Indeed, you need never use a GOSUB statement on the Acorn computers, although it is available for compatibility with other BASICs. The Bubble Sort subroutine could be written like this:

```
1000 DEF PROCbubblesort(N1,N2)
1002 LOCAL NOSWAP, Z$, I
1004 REPEAT
1010 NOSWAP=TRUE
1020 FOR I=N1 TO N2-1
1030 IF A$(I)<=A$(I+1) THEN GOTO
   1080
1040 Z$=A$(I)
1050 A$(I)=A$(I+1)
1060 A$(I+1)=Z$
1070 NOSWAP=FALSE
1080 NEXT I
1090 UNTIL NOSWAP
1100 ENDPROC
```

Again, to sort items 5 to 20 this could be called as follows:

```
100 PROCbubblesort(5,20)
```

Notice how the input parameters—in this case 5 and 20—can be specified in the PROC statement.

Notice also how the temporary variables were declared as LOCAL inside the procedure. This is a way of defining a new variable that limits it to use within the routine itself. It ceases to exist once the routine has been left.

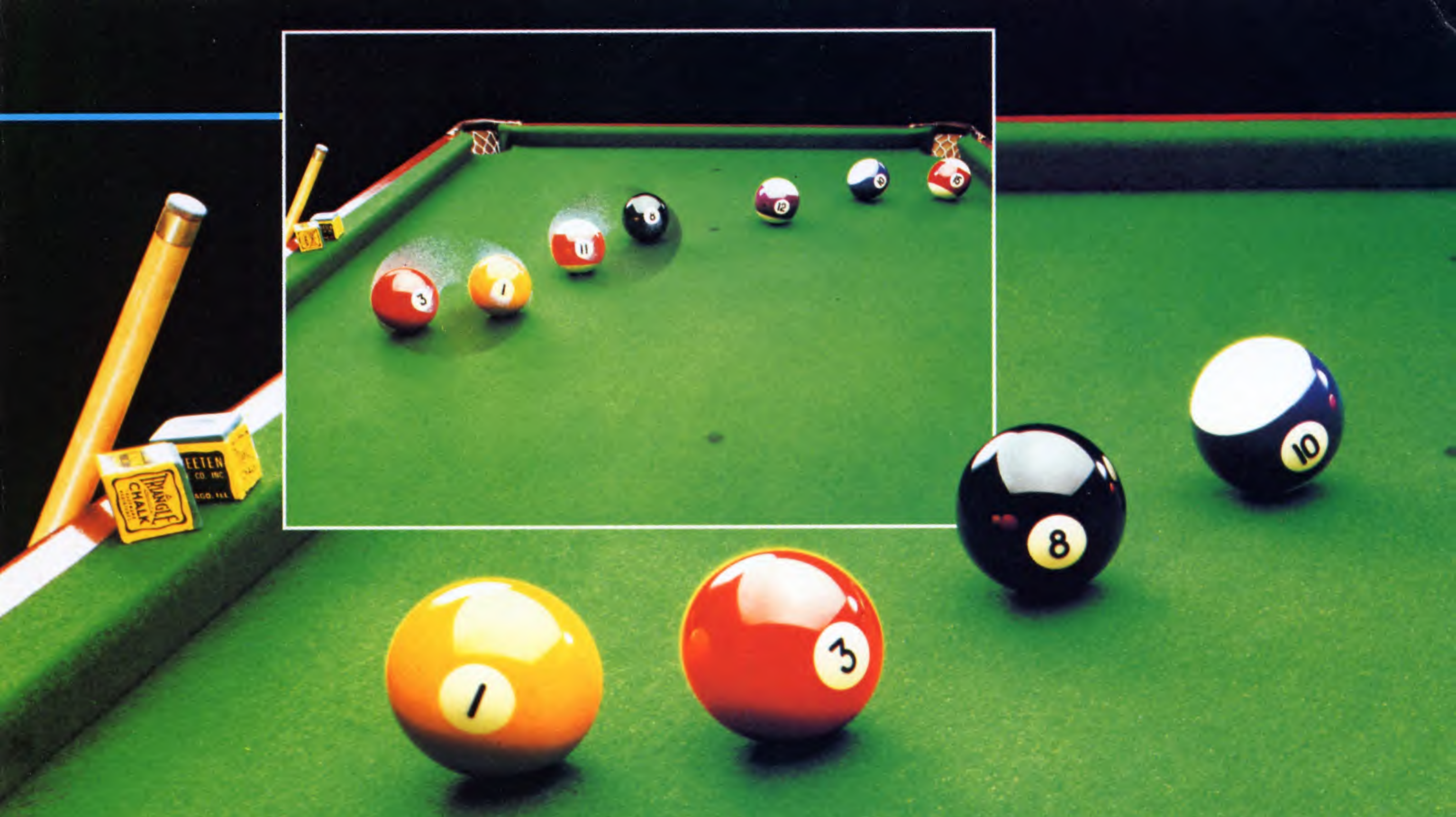
The use of local variables avoids the possibility of corrupting *global* variables—that is, the variables used in the main program. It doesn't matter if a local variable has the same name as a global variable, because they are kept quite separate.

Procedures can be used in place of any subroutine that can be called using the GOSUB statement. However, if only one output variable is needed then a function can be used instead. In the Bubble Sort there were many output variables—in fact, the entire sorted list. But the next example returns only one variable. It tells you the day of the week (from 1 to 7) for a particular date:

```
1000 DEF FNdayofweek(DAY,MONTH,YEAR)
1010 LOCAL M,Y
1020 IF MONTH>2 THEN GOTO 1060
1030 M=MONTH+10
1040 Y=YEAR-1
1050 GOTO 1080
1060 M=MONTH-2
1070 Y=YEAR
1080 =(((26*M-2)DIV10)+DAY+6+Y
   +(Y DIV4)+1)MOD7+1
```

To use this function you will need some extra lines to let you input a particular date and to print out the result:

```
100 INPUT DAY,MONTH,YEAR
200 PRINT FNdayofweek(DAY,MONTH,YEAR)
300 END
```



Note that the year must be input as, for example, 84 not 1984. Of course, you'll want to make a better screen display than this. But Lines like this are useful for testing—which is what you must do next in any case.

TESTING THE MODULES

Each of the modules in your original design should eventually become a subroutine in your program. This method of breaking up the program helps during the testing stage, as each of the modules may then be tested or debugged individually. The idea is to set up the input variables, call the subroutine and then examine the results. Taking the Bubble Sort routine again, you can test it like this:

```
8 INPUT "NUMBER OF ITEMS□";N
10 DIM A$(N)
12 PRINT "INPUT ARRAY ITEMS"
14 FOR I=1 TO N: INPUT A$(I): NEXT I
16 INPUT "RANGE TO BE SORTED ";N1,N2
18 GOSUB 1000
20 PRINT "SORTED LIST:"
22 FOR I=1 TO N: PRINT A$(I) : NEXT I
24 GOTO 16
```

The above program will work on the Dragon but you need an extra command to clear enough memory space:

```
6 CLEAR 1000
```

On the Spectrum change Line 10 to:

```
10 DIM A$(N,10)
```

And on the ZX81 you'll also have to split up the multiple statement Lines and change Line 8 to:

```
8 PRINT "NUMBER OF ITEMS"
9 INPUT N
```

On the Acorn computers change the semicolons in Lines 8 and 16 to a comma and change Line 18 to:

```
18 PROCbubblesort(N1,N2)
```

In a complex program it would be impossible to test every case of input and output. It would literally take thousands of years to run. However, borderline cases should be checked to make sure that the program detects them correctly. For example, the following routine should be checked for input values of 0, 1, 99 and 100.

```
1010 INPUT "ENTER A NUMBER (1-99)";N
1020 IF N<1 OR N>99 THEN GOTO 1010
1030 RETURN
```

(On the Acorn computers, use a comma in Line 1010 instead of the semicolon.)

It is a good idea to make sure that you have run every line of your program at least once during the debugging stage. So all conditional branches such as IF statements should be checked with the condition both true and false.

PUTTING IT ALL TOGETHER

Finally all the modules should be linked together and the program tested as a whole. This is known as *program integration*. If time and care have been taken in the earlier stages then this process should be relatively painless. However, if problems do occur, any suspect modules should be rechecked and altered as necessary. At last, then, you should have a perfectly structured program that does exactly what you want.

So, to refresh your memory, the rules for writing a structured program are:

1. Write a general description of the program.
2. Break this down into modules for as many levels as necessary.
3. For each module draw a flow chart and define the input and output variables and any other effects such as the screen display.
4. Write the programs for each of the modules using the structures described in part 1.
5. Test each of the modules by supplying inputs and checking the results and outputs.
6. Combine all the modules together and test the entire program—it should work!

And in case you'd forgotten, the reasons for going to all this trouble are: readability, testability, changeability, reliability and portability—a lot of ability!

Structured programs are easier for yourself and others to follow. They are easier to debug and modify. They are more likely to work and they are also easier to convert to run on other computers. If you are aiming for any of these then you should use structured techniques.



HOW THE BUBBLE SORT WORKS

The bubble sort program was used to show how a module can be built up, and then tested, before linking it into the main program. Sort routines are very useful in all types of program. This one sorts words into alphabetical order, but it could equally well be used to sort numbers into numerical order. Simply change the string variables Z$ to Z, and the string array A$( ) to A( ).

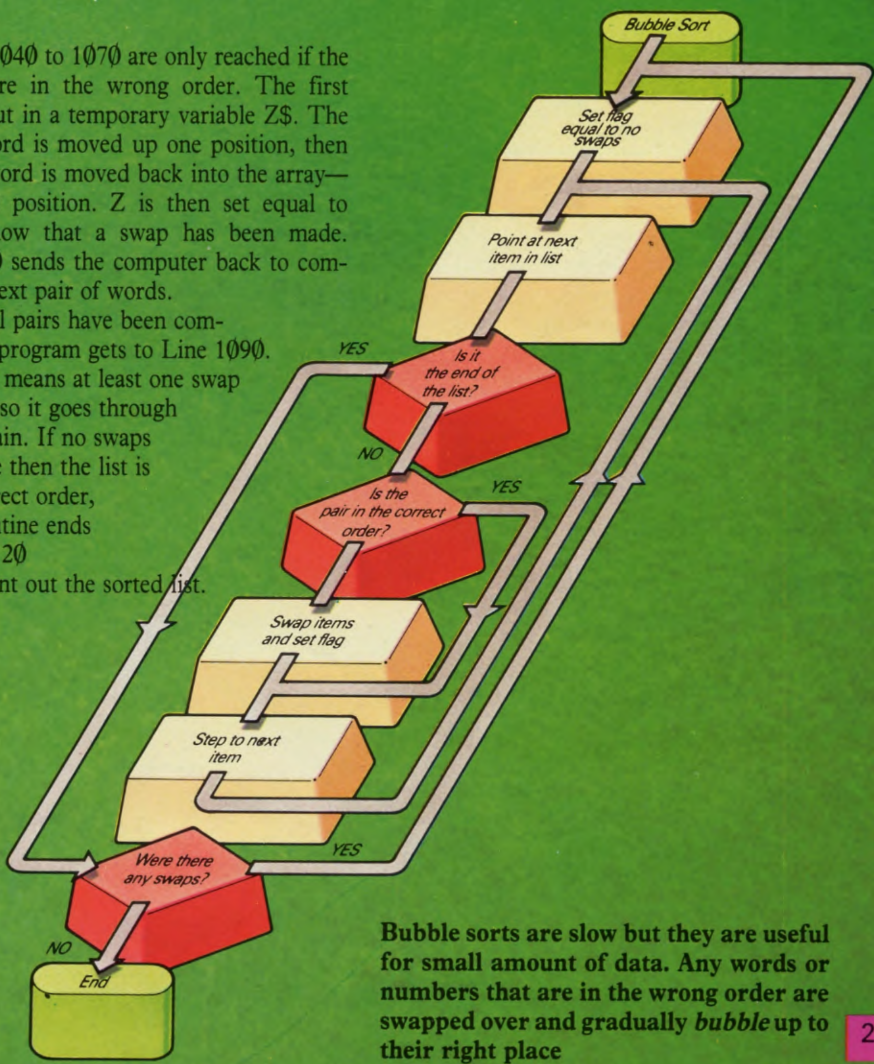
The computer reads through the list comparing pairs of items at a time. If they are in the correct order it leaves them alone. If they are in the wrong order it swaps them round. It keeps going through the list making more swaps until all the items are in the correct order.

To see how the program works in detail it's best to compare it with the flow chart. The first part of the program (not on the chart) finds out the number of items in the list—N—and sets up an array called A$( ) with enough space for N items. Lines 12 and 14 get you to input the words—which are then stored in the array—and Line 16 asks which ones you want sorted. If you want to sort the whole list, type 1, then a comma, then whatever N is. The subroutine itself is called at Line 18.

The routine starts by setting Z equal to zero. Z is known as a *flag* and it records if any swaps have been made. Line 1Ø2Ø creates a loop to run through the list. The numbers make sure each pair is compared once. Line 1Ø3Ø compares the first two words, and if they are in the right order the program jumps over the swap routine and goes on to the next pair.

Lines 1Ø4Ø to 1Ø7Ø are only reached if the words were in the wrong order. The first word is put in a temporary variable Z$. The second word is moved up one position, then the first word is moved back into the array—down one position. Z is then set equal to one to show that a swap has been made. Line 1Ø8Ø sends the computer back to compare the next pair of words.

Once all pairs have been compared the program gets to Line 1Ø9Ø. If Z = 1 it means at least one swap was made so it goes through the list again. If no swaps were made then the list is in the correct order, the subroutine ends and Lines 2Ø and 22 print out the sorted list.



Bubble sorts are slow but they are useful for small amount of data. Any words or numbers that are in the wrong order are swapped over and gradually *bubble* up to their right place

JOYSTICK CONTROLLERS

A joystick is one of the cheapest and most versatile alternative control systems you can buy—whether you just want to enhance your games playing or put it to more serious use.

Although home computers have rapidly grown in sophistication, one thing they still cannot do is to give themselves instructions on what to do next. Telling the computer what to do will always be the responsibility of the user. And learning how to communicate with the computer—or how to write programs that make it easy to do so—is part of what makes using a computer so interesting and so instructive.

Although computer designers are hard at work on alternatives, the main method of communicating with the computer is still one that dates back to a time when computing was in its infancy—the keyboard. There are many reasons why this is less than ideal—not least the fact that it is a rather slow and laborious system, and that you have to learn typing skills along with learning to use the computer.

For the moment, if you want to write programs, then there is no real alternative to the keyboard. But this is not the only thing that the keyboard is used for. You have already seen how to write programs that ask the user to communicate instructions to the computer (pages 129 to 135). Getting the computer to act on user responses is a feature of everything from arcade games to business programs. And once again, the way it receives these responses is through the keys.

But there are two types of response here. On the one hand there is the verbal or numerical response, where the computer asks you to INPUT something like 'Name and date of birth', and expects a reply like 'Harold Smith 9.7.53'. The second type of response is an analogue, where given keypresses do not produce the letter that they are normally assigned, but have been given some other quite different function—such as pressing an X to move a cursor to the right and a Z to move it to the left. You have seen how to program your computer for both of these kinds of response in the articles on page 129 and on page 54.

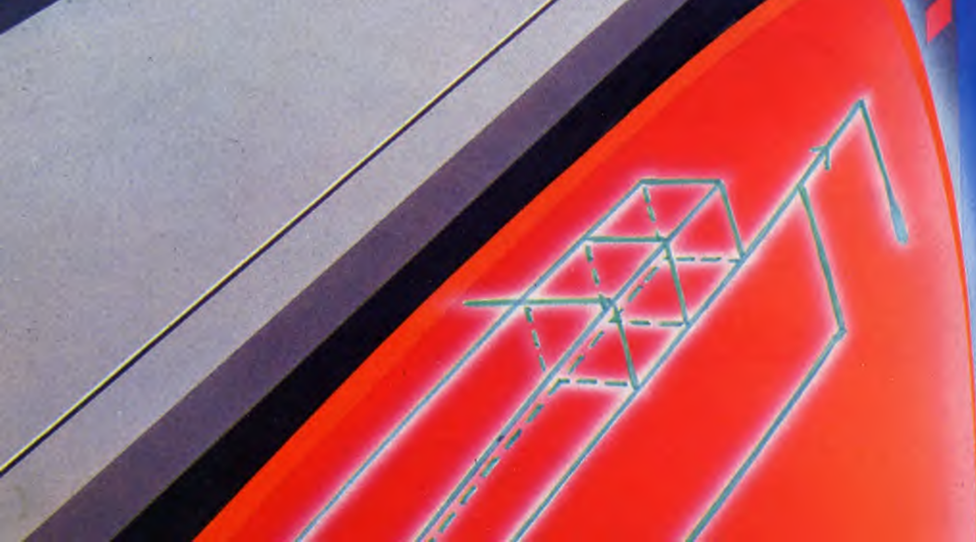
The first kind of response is obviously tied very closely to the keyboard and the need to be able to give the computer precisely different letters and numbers. But the second type has no such obvious link, and it is hardly very user-friendly programming to expect someone unfamiliar with the keyboard to tie their fingers in knots trying to remember where the P that they need to press to move up is, at the same time as hitting F to fire the gun!

Fortunately, there is an alternative. Joysticks for popular home computers are an inexpensive peripheral that open up whole new areas of control and communication. They are perhaps unfairly associated with games playing only—and while this is their most familiar use, it need not be their sole function.

With proper programming, even the humblest of arcade-game type joysticks can take over many of the roles traditionally reserved for the keyboard, and so point the way to the future, with real user-friendliness.

For example, in a graphics program, it is perfectly possible to use a joystick or similar control to guide a cursor that draws on the screen—a program to do this under keyboard control is on page 132—or even selects a colour from a 'paintbox' displayed on the screen. A joystick could even be used to take over some of the functions of the keyboard in giving verbal responses, with a little bit of rethinking on the part of the programmer. An example of this is a feature of many arcade games, which have a 'hall of fame' displaying players names and their high scores. The names—a verbal response—are actually INPUT by using the joystick to scan across the letters of the alphabet, then 'firing' to select the correct ones. With a little imagination on the part of the programmer, it would be easy to incorporate this into that 'Name and date of birth' routine, or at another level, offer the choice between a menu of program options, say.

So much for the potential of one of the cheapest and most versatile peripherals. In a future article, you will see how to program your computer to operate under joystick control, ready to push back the boundaries of user-friendly programming—or to add to the enjoyment of arcade games! But first, a look at the hardware that's available to you.



TYPES OF JOYSTICK

Not all joysticks are suitable for every computer. You will need to ensure that whatever you buy is compatible for your machine—as covered later in this article. There is a further restriction, and that is that if you buy commercial software, on some machines you need to ensure that it is written for the joystick you buy. But within these basic limitations, there is a number of choices.

- COMMUNICATING WITH THE COMPUTER
- USING A JOYSTICK
- JOYSTICK, TRACKER BALLS AND MICE
- HOW JOYSTICKS WORK
- ANALOGUE OR DIGITAL
- CONTROLLING THE COMPUTER
- CHOOSING THE RIGHT JOYSTICK FOR YOUR COMPUTER

The simplest type of joystick has a case with a stick which you can move to left and right, up and down and diagonally, plus a button—the 'fire' button, although it can actually be used for anything from making a games character jump, to selecting a letter or dropping a bomb.

For a price, there is plenty of elaboration on this basic theme. For example, some joysticks have multiple fire buttons. These can simply allow more comfortable play, or aid left-handed players. With suitably-written programs, they might also be given different functions—such as one to shoot and one to drop a bomb, or one to file and one to edit, for example. Some models are specially ('ergonomically') shaped to fit the hand, giving a firmer, more comfortable grip, and more precise control. One style is shaped as a pistol grip, complete with trigger in some cases.

But the joystick family includes some devices where the latest designs do not even look like joy sticks. For example, there are pointers that use gravity to detect small movements in the user's hand as he or she indicates a screen location. These pointers use mercury switches. As the stick is tilted the mercury tips to one end of the switch and so makes or breaks contact. They are extremely sensitive—often too sensitive for games unless the software is specially written. There are also touch-sensitive pads that analyze the finger pressure of the user to indicate a direction.

Yet another type is the tracker ball. Originally developed for military and avionics applications, these were soon adopted for commercial arcade games machines, and are now finding their way into the home computer market. Control is by means of a ball which protrudes above the surface of the case, which can be rolled in any direction with the fingertips.

At their most sophisticated, such control devices lead on to systems designed for professional and business use. A feature of many of the latest business computers is a 'mouse'. This is very like an upside-down tracker ball in some cases, although there are also versions that work on other systems. The mouse works by rolling the device over the surface of the desk or a pad. Movements of the mouse are translated into movements on the screen. It is the text cursor that's moved and the mouse really takes over operation of the cursor keys. They allow quick movement around the screen to alter data, say, or to select items from a menu. Selections can be made with buttons on



the top of the case. Unlike normal joysticks they are not generally used for drawing on the screen or for controlling movement in games.

Graphics tablets are for drawing. Sensors built into the 'pen', or the pad itself allow realistic direct 'drawing' for commercial graphics applications, although they are also becoming available for domestic use. Light pens allow direct 'drawing' on the screen, and are also popular for home computers.

HOW THEY WORK

Although they may seem widely different at first sight, all of these devices in fact work in a similar way. In all of them, movement is translated into a changing pattern of electrical signals that can be 'read' by the computer.

As with all electronic devices, joysticks can be either digital or analogue—although your computer will only be compatible with one or the other—see below. In the digital type, there are a number of electronic 'switches' which are opened or closed to form a unique pattern of binary digits that is dependent on the position of the control. Analogue types contain potentiometers (variable resistors). Movement changes the voltage passing through these to give a unique combination for each position of the control.

In appearance, the two types of simple joystick are usually easy to differentiate. In the analogue type, there are two potentiometers set at right-angles and operated through a mechanical linkage. The stick itself, as a result, is not centre-balanced, but remains in whatever position it is left by the user. Conversely, digital types are usually centre-balanced, and when released, spring back automatically to the central, or neutral, position. This is not an absolutely fool-proof test, however, since there are a few centre-balanced potentiometer types.

Although you have little choice in the matter of which type you buy, because of the compatibility problem, there is a distinct difference between the feel of these two systems. Centre-balanced types are stiff and usually only move a small distance. This adds a certain realism, because greater effort is met by greater resistance, but it does have the disadvantage of tiring your hands. However, the effect of either type on the computer is controlled more by the program than the joystick itself. Generally, a well-written program can make the joystick responsive and easy to control. For example, the program determines whether a slight movement of the stick away from the neutral position causes a slow, slight movement on the screen, or a fast, continuous one.

Touch pads consist of a pair of plastic sheets, held a slight distance away from one another. On these sheets are laid down a fine grid of resistances or conductive paths, and these are oriented at right angles to one another between the two sheets. Touching the pad at any point lays down a unique voltage pattern that can be scanned by the computer. Some graphics tablets also work like this on a larger scale although there are several different systems. However, it can prove difficult to manufacture a uniformly-sensitive touch pad. A disadvantage in use is that if the surface gets soiled—finger patches are common—movement may become erratic due to skidding across the pad.



Tracker balls offer the advantages of being extremely light to operate, because the ball has no direct mechanical connection to the sensing system. It is supported on rollers which are free to turn in two directions at right angles to one another. As the ball is spun, in whatever direction, it turns one or both of these rollers by a certain amount. The rollers are then either connected to potentiometers or to a digital sensor—one system uses a rotating disc to interrupt the beam of light from an LED falling on a phototransistor, then counts the pulses. Whichever system is used, once again, the computer is able to interpret the electrical signals in terms of a particular pattern of movement on the screen.



TROUBLESHOOTER

Joysticks can sometimes be the cause of annoying 'crashes' or LOADing problems that are all too easily blamed on the software or on the cassette recorder. Make sure that your joystick is *securely* plugged in before you try to LOAD the tape—and on the Spectrum check the security of *both* the interface and the joystick connections. Also on the Spectrum, check for compatibility with the joystick that you are using—see main text.

OPERATING THE COMPUTER

On page 132, you saw how a program can be written to allow control through pressing certain keys for certain functions. This is very simple to do, using the BASIC commands GET$ or INKEY$, which scan the keyboard, looking for a particular signal which indicates that one or other of the keys has been pressed. If the computer finds a particular keypress, then it will carry out a further operation as instructed by the program—such as moving a missile base to the right if an X is pressed.

Essentially, the operation of the joystick is similar to this. However, the keyboard is an integral part of the computer, while the joystick is not. So the joystick has first to be connected to the computer through a suitable port, or series of terminals, by which the computer can communicate with the outside world. Then the computer has to be programmed to scan those terminals, looking for a particular signal which means that the joystick has been moved to the right, say.

This is relatively straightforward, and in a later article in the series you will see how to write a program which will do this, transferring control completely to the joystick. But you can see why this raises problems of compatibility. Firstly, the joystick itself must be able to connect to the computer's port—either directly or through some type of interface. Then, the programmer must know what signals the joystick will present at which terminals, or it will not be possible to program the computer to look for these.

The results of such a system could be impossibly confusing, but thankfully there are some standards which different makers follow to a greater or lesser extent, and for each micro, there is a definite set of rules. The commonest standard, which has become virtually universal, is the Atari joystick, which is a centre-balanced type. Many manufacturers offer an Atari-compatible joystick, and many computers are designed to accept these joysticks.

To begin with, the Commodore 64, Vic 20 and the Spectrum work with centre-balanced (digital) type controls. The BBC, Dragon and Tandy use non-centre-balanced (potentiometer) types—although some centre-balanced potentiometer type sticks are available. But within this, there are still important individual differences between the machines in any group. The most important of these concern the interfacing.

INTERFACING

As with any computer peripheral, joysticks need to be interfaced with the machine. The interface may be part of the computer's hardware, or it may be a separate device.

Where you are able to plug a joystick straight in, manufacturers commonly incorporate an Atari-type interface into the computer, which will then accept any joystick which is an Atari-compatible model.

The other choice is to provide one or more analogue ports, which are compatible with potentiometer-type sticks. This is a little less common, because it means that two analogue-to-digital converters have to be built in for each joystick that is to be used.

Choosing a joystick for a Spectrum micro is far more difficult than for the other machines, as the choice is the widest of all. There are so many very popular Spectrum joysticks, that some commercial software even has a menu before you start, allowing you to choose which of maybe four or more models of joystick you wish to use with the game.

The differences in fact lie not so much with the joysticks themselves as with the interfaces—because Sinclair do not provide a joystick interface with these computers. Before a joystick can be used, an interface has to be connected to the edge connector that protrudes from the back of the machine. Spectrum owners have to purchase the interface as well, which approximately doubles the cost.

The interface is usually housed in a separate box which sits at the back of the machine in direct contact with the terminals of the edge connector in the user port. There is also a type which is built into the case of the joystick itself, with a separate connector on the lead, but these are far less common.

Some of the Spectrum joysticks are Atari-compatible, but different joystick standards address different terminals in the user port. The software looks at particular terminals for the incoming information from the joystick interface, so unless the correct standards are used, the Spectrum will be looking in the wrong place for the joystick information.

This leaves a problem for Spectrum owners with a big collection of commercial software—in that not all of it may be written to suit a particular joystick standard. It's a good idea to check all the software you own, or want to buy, before you purchase a joystick, and make a list of the standards that are compatible with your software. If you are lucky, there may be a universally compatible type in the list, but it is just as likely that there will not be. If there is no common type, you have four choices.

The first is to forget about using some of your software with a joystick at all, which is clearly a very poor choice, but the only way in which you will avoid having to spend more money than you perhaps bargained for. There is an alternative, though it is not a conventional joystick—a mechanical stick which fits over the keys and operates them directly.

Secondly, you can buy two or more interfaces. This can prove to be very expensive.

Thirdly, you can buy a programmable interface. This is much more expensive than the ordinary type—at least the price of a data cassette recorder, say. But for the money you do at least get a device that allows you to use your joystick with any software—even with games that do not offer a joystick option. These devices work by addressing the ports used by the keyboard and allow you to define which keys will be activated by the joystick. This gives the user the capability to fool the Spectrum into believing that the joystick movements are, in fact, keypresses. The main disadvantage of the programmable interface is that you need to reprogram it every time you use it with a different standard.

The fourth option is to purchase compatibility cassettes which allow the use of different joystick standards. These are about the price of a game, and effectively allow you to add a joystick option to any commercial software. Their greatest disadvantage is that you have to load two cassettes every time you want to use your software.

Sinclair's own interface, the Interface 2, allows you to connect two joysticks at once, so letting two players play against each other when the software is written to include this option. It also allows the use of ROM-based software cartridges, which have recently become available to Spectrum users. Unfortunately, there is currently very little software that is compatible with this interface, so you may find your favourite game is unsuitable.

The ZX81 is far from the ideal games machine. However, like the Spectrum, it can be interfaced for a joystick—and the operation of the interfaces is similar to those described above. But the cost of these may well be almost as high as the cost of the computer itself.

The Commodore 64 and the Vic 20 both have a pair of digital joystick sockets. This allows two players to compete with each other when the program is written to make use of this facility. Since the sockets are both connected through an Atari-compatible interface, this means that Commodore users have plenty of choice between the joysticks of this type on offer from various makers.

The BBC computer uses non-centre-balanced potentiometer-type joysticks which plug directly into the analogue port. The Dragon and Tandy are similar, but the joysticks which fit these machines are not suitable for the BBC computers, as the manufacturers have chosen different plugs to go with their joysticks. There is little choice for Dragon owners, although Dragon Data do make their own stick. Electron owners can buy special interfaces compatible with Atari-type sticks.

BUYING A JOYSTICK

The first, and most important, thing is to make sure that whatever you buy is compatible with your computer—and, for Spectrum users, with your software as well.

Then think about how much you want to spend—and again if you are a Spectrum owner, bear in mind that the interface may well cost as much as the joystick, or even more. The simplest sort of joystick costs about the same as a game, while the more elaborate types can cost twice and three times as much. At the opposite end of the scale, for about six times the cost of the cheapest joystick, you can afford the cheapest tracker ball—although the most expensive can cost as much as the computer!

Where possible, if you are buying from a dealer and not by mail order, ask for a demonstration. Comfort and convenience are very subjective things, and what might be one person's ideal joystick might not suit someone else at all. Remember that if you are keen on games the joystick might well be subjected to hours of gruelling use.

Q+A

Why are there so many different types of alternative control system?

Different designers and makers have tried out a large number of alternatives to keyboard control, systems which allow the user to 'point' in some way or another to give the computer its instructions. But as yet, many of these devices are not widely available and too few people have tried enough different types for there to be any long-term evaluation of the merits of one design over another.

Also, most of the devices were developed for specific applications and to meet particular needs, such as the graphics tablet of the computer graphics industry, or the joystick and cursor disk on some word-processors, for example. The use of the equipment has then often diversified into other areas where it is more of an unknown quantity.

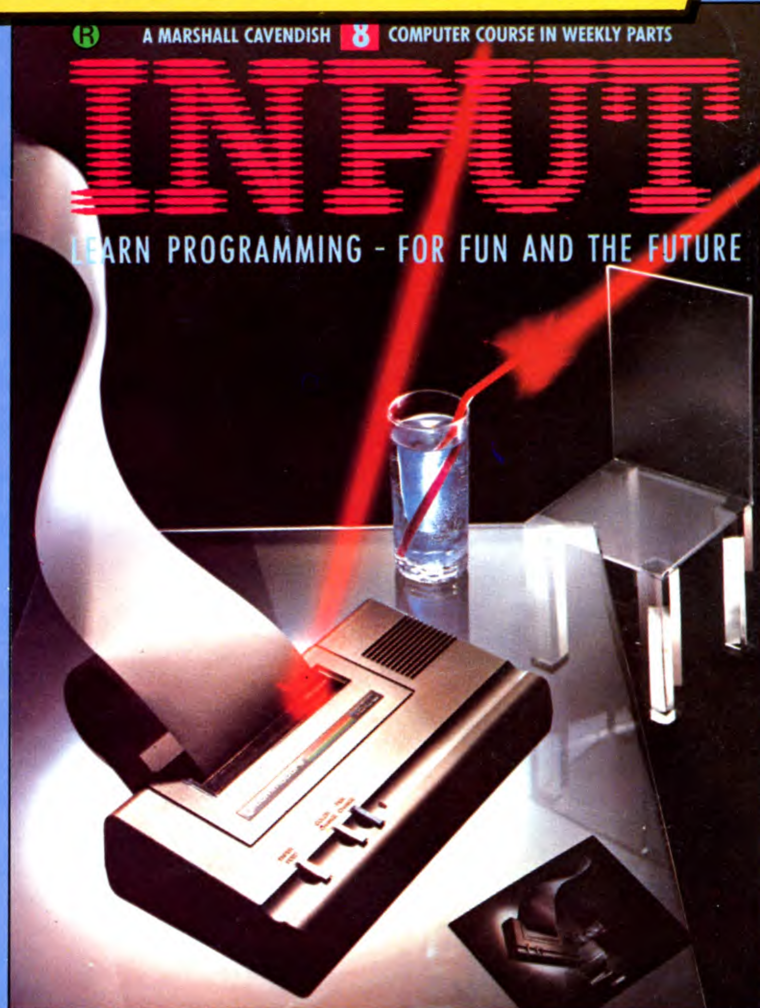
Generally, for home use, the simple joystick has so far proved to have the best balance of performance against cost—although the balance is shifting, to see some of the other types becoming much cheaper and more freely available.

CUMULATIVE INDEX

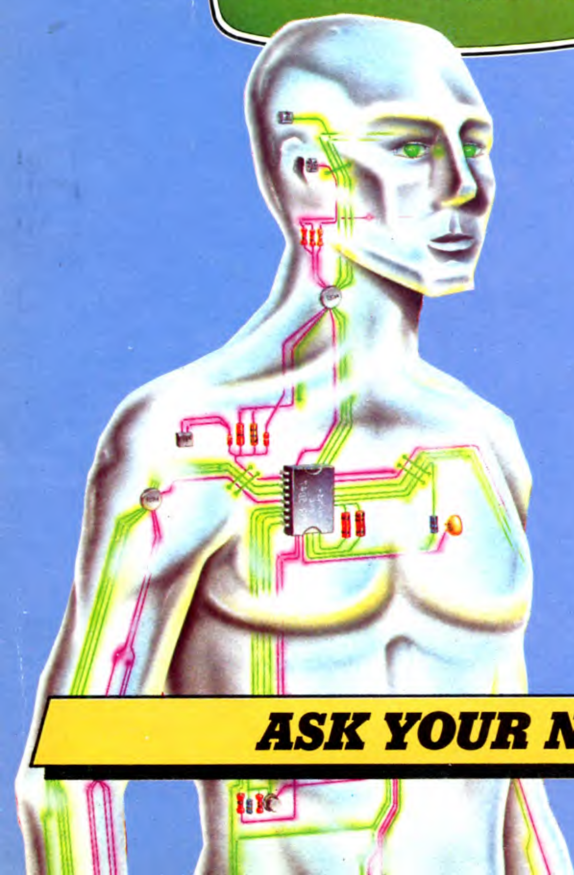
An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

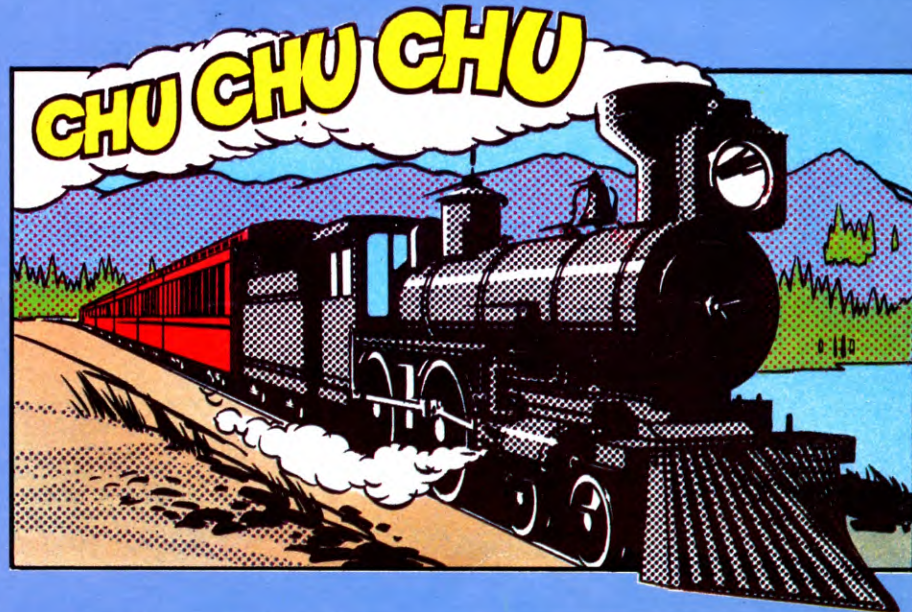
COMING IN ISSUE 8 ...

- *The MICROPROCESSOR controls your computer, but how does it work?*
- *Brighten up your Dragon or Tandy with some COLOUR UDGs*
- *See how SIN and COS can improve your drawings*
- *Learn how to get the most from your computer with PEEK and POKE*
- *Add some SOUND EFFECTS and liven up your games routines*
- *Find out all about PRINTERS and what each type can do*









ASK YOUR NEWSAGENT FOR INPUT